Introduction to Software Engineering

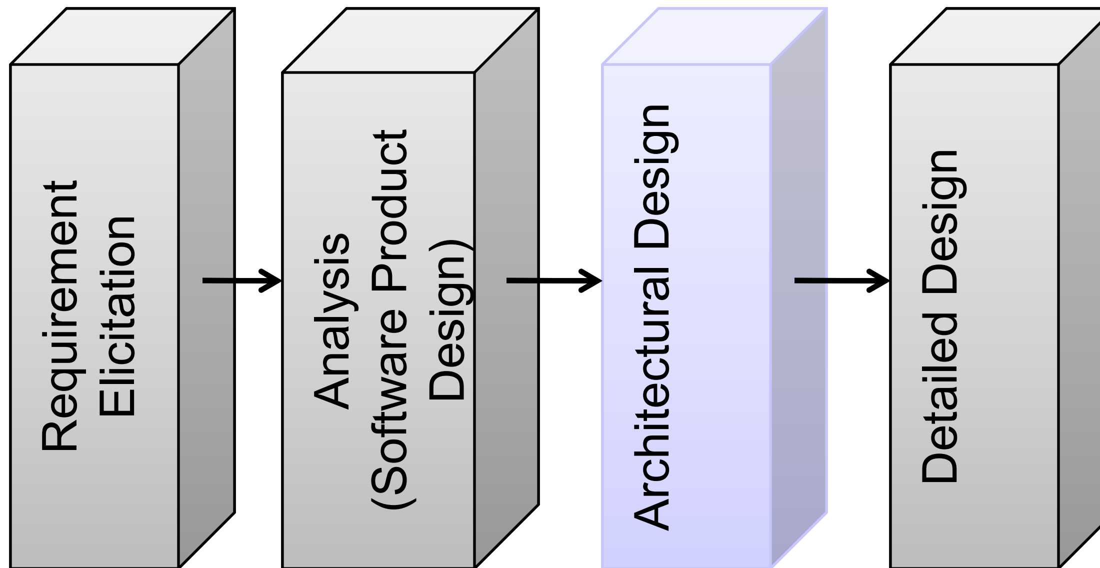
ECSE-321

Unit 8 – Architectural Design

Before Architectural Design

- Requirements Elicitation
  - Understand the user's needs, desires
- Analysis (Software Product Design)
  - Structure and formalize the requirements
  - Produce a model of the system
    - From user's perspective
    - Still ***NO*** implementation-specific information

Architectural Design

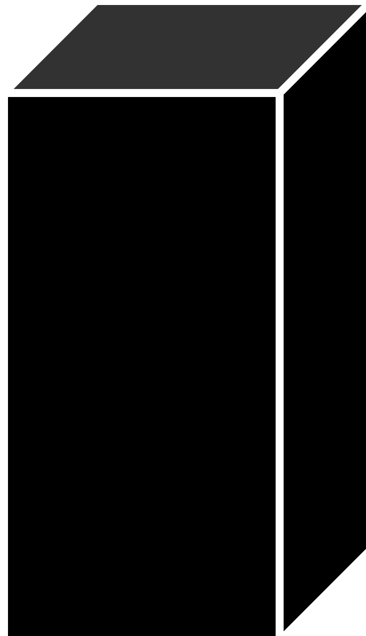


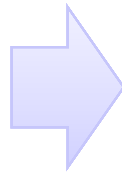
In Software Product Design

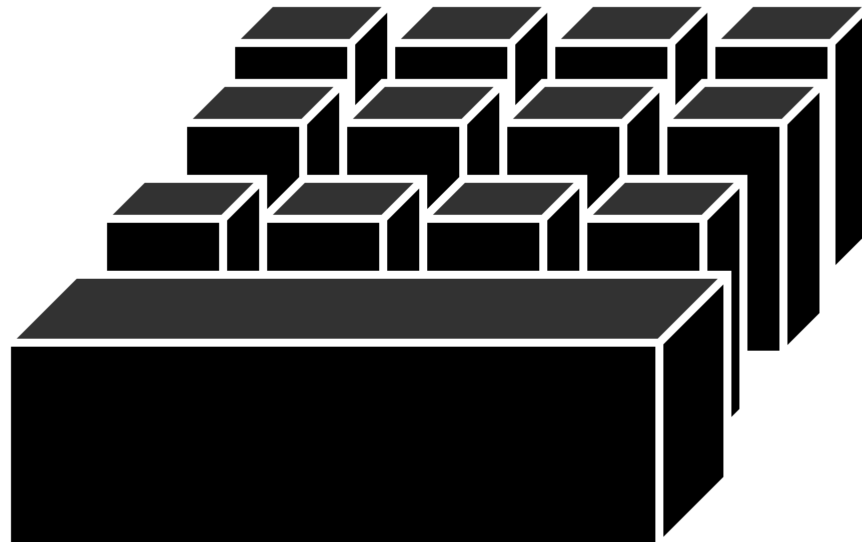
- We treated the software as a “black box”
  - refined the user needs to arrive a software requirements specification
  - only external form and behaviours were specified
  - details that can go inside the black box were left out

Architectural Design

Higher level structure among the constituent black boxes is established during architectural design







@ Software product design

Each box is further refined during detailed design

Architectural Design

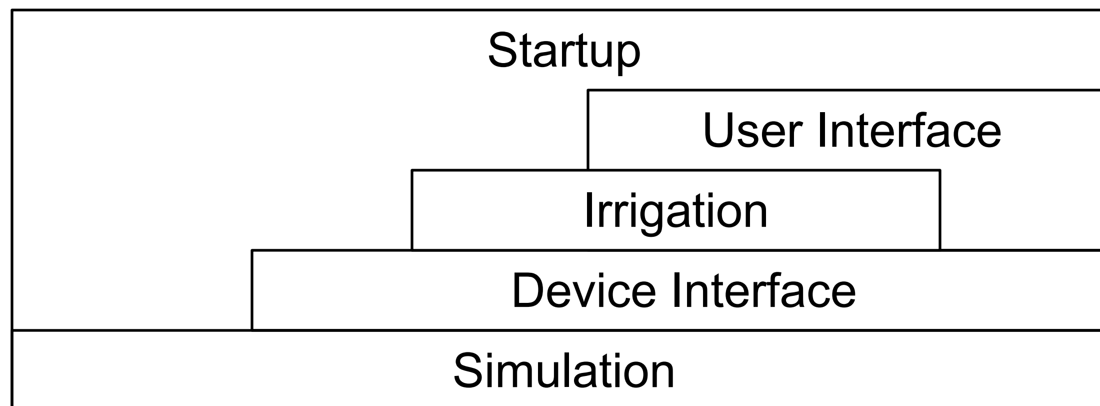
- Architectural design is the activity of specifying a program's major parts
  - responsibilities of the major parts
  - properties of the major parts
  - interfaces of the major parts
  - relationships among the major parts
  - interactions among the major parts
- Goal: determine the high-level structure – or ***software architecture***

Detailed Design

- Detailed Design is the activity of specifying the internal elements of all major parts
  - structure of major parts
  - relationships
  - processing including algorithms and data structures

Architectural Design Specification

- Following items appear in the software architecture specification:
- ***Decomposition*** – extent depends on the size of the product



Modules
of a automated water
irrigation control system

Architectural Design Specfications

- ***Responsibilities***
  - each module must be in charge of certain data and activities
  - “startup” module above configures the system the irrigator at power up
  - “user interface” is responsible for tracking the state of the user interface and changing it to respond to user actions and system state
  - “irrigation” module tracks the state of the valve, sensors, and controls irrigation
  - “device interface” module provides a unified way of connecting to a multitude of devices
  - “simulation” provides a simulation mode for evaluating different irrigation schemes

Architectural Design Specifications

- ***Interfaces*** - boundary across which modules communicate
- Interface specification – description of the mechanisms used for communication over an interface
- Important to have interface specifications for modules – so they can be developed independently

Architectural Design Specifications

- Collaborations
  - modules collaborate to achieve their processing goals
  - for example, in the irrigator example, the “user interface” module is responsible for displaying the progress of irrigation. The irrigation module needs to update its state to the “user interface” and also the “user interface” may want to pull information from “irrigation”.

Architectural Design Specifications

- Relationships
  - architectural design can impose restrictions on the ways modules communicate
  - for example, with “layered” modules, only adjacent modules can communicate.
  - The activities in a module can depend only on modules directly below/above it.
  - A module can impact a module that is only directly above/below it.

Architectural Design Specifications

- Properties
  - Modules can properties such as timing restrictions and resource usage restrictions
  - For example, we can say “simulation” module should complete a given simulation in 10 seconds. Additionally, there could be a memory restriction on the “simulation” module as well.

Architectural Design Specifications

- States and State Transitions
  - modules have important states that affect their externally observable behaviour
  - these states should be specified as part of the architectural design
  - for example, the irrigator can have two modes: manual and automatic. the “irrigator” module provides this state but is should be available to the “user interface” module.

Architectural Design - Summary

- How to remember all of what goes on in architectural design?
- DeSCRIPTR: Decomposition, States, Collaborations, Responsbilities, Interfaces, Properties, Transitions, and Relationships
- Architectural design ***need not always include*** all aspects

Detailed Design Specifications

- Goal: fill in the details left after the architectural design
  - complete the specifications so that programmers can implement it
- Detailed design includes all the activities of architectural design at a finer grain and more
- DeSCRIPTR + ??

Detailed Design Specifications

- Packaging and implementation
  - programs can be divided into packages (containers)
  - allocation of program units to packages and their visibilities may be specified
  - For example, classes in each package and their visibility can be specified as part of this activity

Detailed Design Specifications

- Algorithms, Data Structures, and Types
  - detailed design is least abstract – however, not yet code!
  - key data structures and algorithms may be specified as part of the detailed design
  - aspects that can significantly impact performance or usability can be selected for this specification
- PAID – Packaging, Algorithms, Implementation, and Data Structure & Types

Design Principles

- During architectural + detailed design, many alternatives could be generated and evaluated
- How to evaluate the alternate designs?
- Use design principles as a way of measuring "goodness" of potential candidates

Design Principles

- Software engineering shares with other design approaches
  - high quality products
  - long-lived products
  - products meeting stakeholder needs
  - not costing too much

Basic Design Principles

- Principles stating the characteristics that make a design better able to meet the stakeholder needs → basic design principles

Principle of Feasibility

A design is acceptable only if it can be realized

- Design must specify something that can be built and will work!
- An unrealizable design is not acceptable.

Principle of Adequacy

Designs that meet more stakeholder needs and desires, subject to constraints, are better

- It may not be possible to meet all requirements – meet as many (and important) requirements

Principle of Economy

Designs that can be built with less money, in less time, with less risk, are better.

- Software development is expensive, time consuming, and risky
- Good design can specify a program that can be built, tested, and deployed on time within budget.

Principle of Changeability

Designs that make a program easier to change are better.

- Approximately 70% of lifetime software product cost come from maintenance.
- Change is important and impacts cumulative life cycle cost!

Constructive Design Principles

- Basic design principles
  - provide desirable engineering design characteristics based on stakeholder needs
- Constructive design principles
  - provide desirable engineering design characteristics based on past software development/engineering experience

Constructive Design Principles

- Modularity principles
  - high quality programs are constructed from self-contained, understandable modules with well defined interfaces
  - modularity principles state criteria for judging whether designs specify good modules
- Implementability principles
  - how easy is a design to build?
  - easiness affects both adequacy and economy
- Aesthetic principles
  - beauty is an important design principle – contributes to design quality

Modularity

A modular program is composed of well-defined, conceptually simple, and independent units that communicate through well-defined interfaces.

- What is a module?
  - a design can be hierarchically decomposed into increasingly smaller parts
  - at each level of the hierarchy we can specify a module along with its immediate parts

Advantages of Modularity

- Modular programs are easier to
  - understand and explain
  - document
  - change because modifications are usually restricted to parts within the module
  - test and debug because they can be dealt in isolation. debugging is less likely to introduce bugs outside a module.
  - reuse because a module is likely to have a solution that can be used for other problems

Advantages of Modularity

- Modular programs are easier to
  - tune for better performance because all relevant data and processing are locally available
- Modularity is one of most important characteristics of a well designed program

Modularity Principles

- Design principles that serve as guides and evaluative criteria in forming modules

Principle of Small Modules

Designs with small modules are better.

- Large modules are hard to understand, explain, document, write, test, debug, change, and reuse
- What counts as "small" module?
  - How many immediate parts does a module have?
  - Depends on the complexity that can be part of the module due to the parts

Principle of Information Hiding

Information hiding is shielding the internal details of a module's structure and processing from other module.

- First proposed in 1972 by David Parnas
- Two major advantages:
  - hiding prevents modification of module internals from outside. Protects the module from errant external code and makes module repair, reuse easier
  - module can be used without understanding its internals, reducing complexity, making programming much easier.

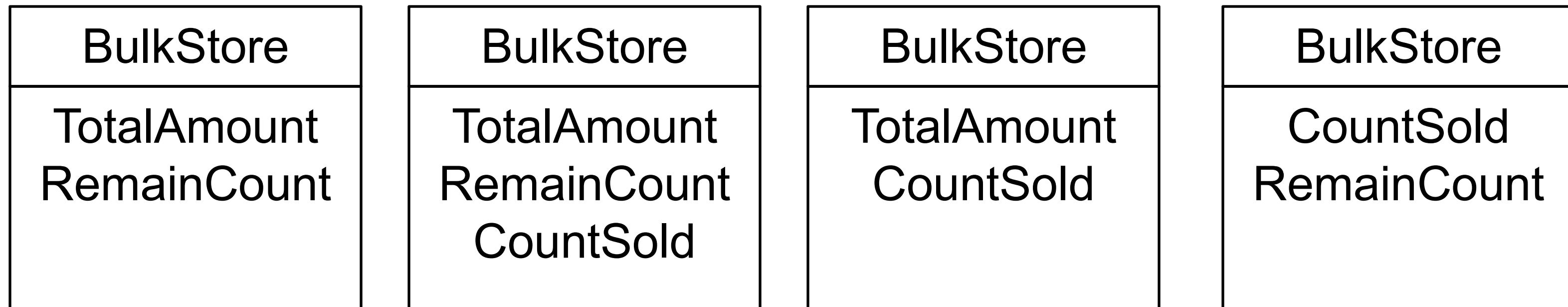
Principle of Information Hiding

- Specific examples of what to hide
  - algorithms – e.g., choices of alternatives for sorting, searching, etc
  - internal data representation – e.g., types and data structures
  - volatile design decisions – e.g., sizes, capacities, waiting times, etc.
  - non-portable code and data – code specific to a certain OS, data such as file paths
  - internal organization – e.g., control flow
  - use of data and operations from other modules

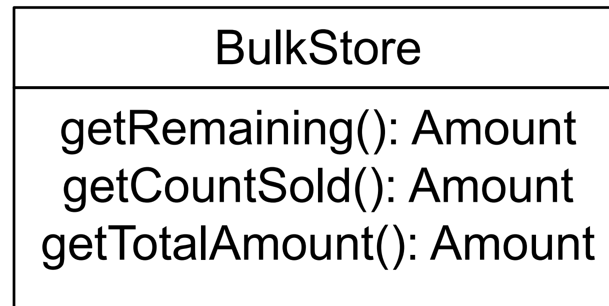
Principle of Information Hiding

- What must not be hidden?
  - name and attributes of data provided – e.g., variables and constants provided by the module
  - names, parameters, and return types of operations (methods) available for clients
  - module error and exception conditions and behaviour
  - other modules with which a module interacts as a client
  - behaviour of operations provided by a module

Principle of Information Hiding



Due to the invariant TotalAmount = RemainCount + CountSold all four implementations are equivalent. Information hiding prevents such details from being exposed to other “client” modules



Principle of Least Privileges

Modules should not have access to unneeded resource.

- Complements the principle of information hiding – restricts access to “internal” processing
- A module having access to unneeded resources is
  - harder to understand
  - less reuseable
  - ** can have serious security problems **

Principle of Least Privileges

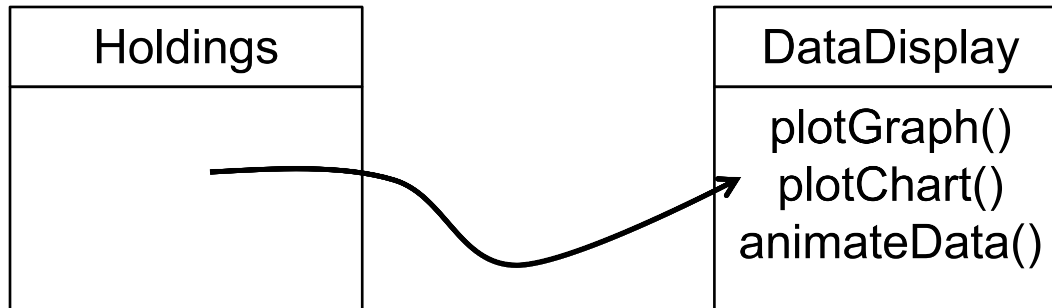
- Example violations of principle of least privileges:
  - importing packages or modules not needed
  - modules with unneeded access to files, databases, or computers
  - classes with references to objects never accessed
  - operations with parameters for data they don't need

Principle of Coupling

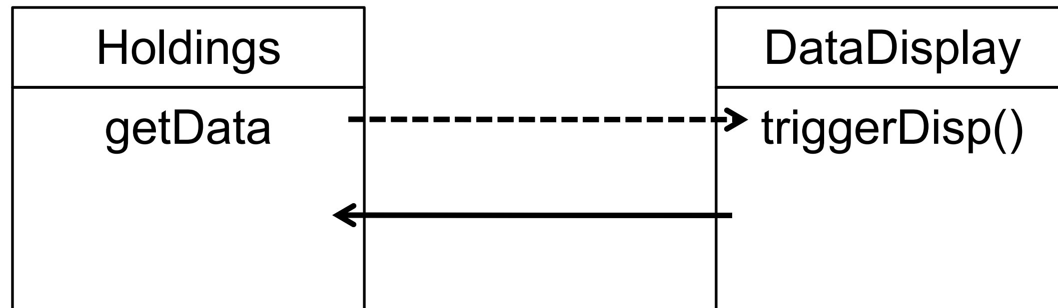
Coupling is the degree of connection between pair of modules.

- Coupling between two modules depend on the amount of communication
  - strongly or tightly coupled – communicate extensively or in undesirable ways
  - weakly coupled – communicate little
  - decoupled – not communicate at all

Principle of Coupling



directly use the methods provided by DataDisplay in Holdings



Holding is independent of the data display process. Just triggers a data display routine.

Holdings trigger DataDisplay to redisplay. DataDisplay pulls the data from Holdings using getData and displays

Principle of Coupling

Module coupling should be minimized.

- Strongly coupled modules are hard to
  - change independently
  - hard to understand
  - hard to document
  - hard to test, debug, and maintain

Factors leading to Tight Coupling

- Failure to hide information; access to internal processing leads to tight coupling
- Modules communicating via global variables instead of direct communication
- Modules communicating via special data types or structures – use simple/standard data types
- If public interfaces are used, coupling is proportional to the number of messages

How to Evaluate Coupling Strength?

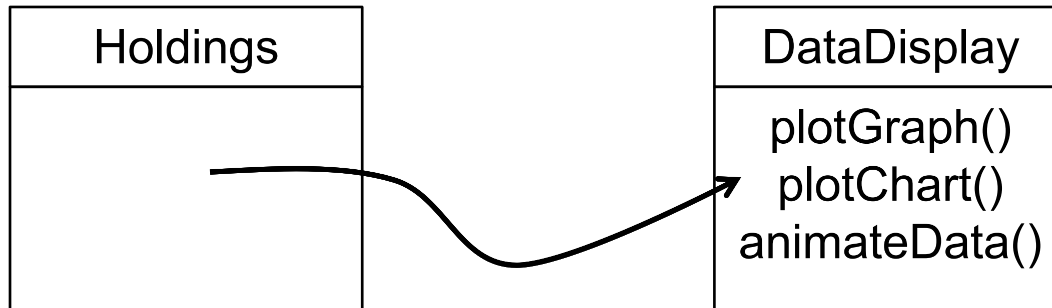
- Can I use each module in some other program without the other?
  - yes – the modules are decoupled
  - yes, but need a substitute for some services provided by the other module – weekly coupled
  - no, at least one module cannot be used without the other – tightly coupled

Principle of Cohesion

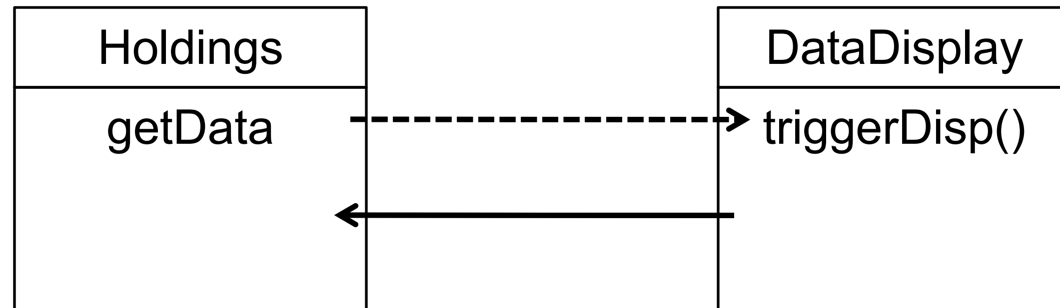
Cohesion is the degree to which a module's parts are related to one another.

- Cohesive modules hold data on a single topic or related topics
- A modules whose parts are strongly related is *cohesive* or has *high cohesion*
- A module without strongly related parts is *non-cohesive* or has *low cohesion*

Principle of Cohesion



Holdings directly deals with data display – maintains state information; not related to Holdings main responsibility – ***low cohesion***



Holdings have very little information about display – ***high cohesion***.

Principle of Cohesion

Module cohesion should be maximized.

- Highly cohesive module should do most of its work by itself
  - Coupling should be minimized

What determines Cohesion?

- Cohesion is high in modules with a single mission
- One way of creating modules with high cohesion is to build modules that implement data types
- Creating objects that relate to physical objects (e.g., thermostat) is another way of achieving high cohesion

Implementability Principles

- Software development is costly, risky, and time consuming
- Implementability principles help with
  - design economy
  - providing criteria for judging whether design could be cheaply, quickly, or successfully built
- Three implementability principles:
  - simplicity
  - design with reuse
  - design for reuse

Principle of Simplicity

Simpler designs are better.

- Simpler designs are created, documented, coded, tested, and debugged faster
- Efficiency and reusability can lead to complex designs – need to carefully evaluated

Principle of Design with Reuse

Software reuse is the use of existing artifacts to build new software products.

- Items reused during software development
  - project requirements
  - product requirement
  - design elements
  - code units
  - test plans & cases
- Reuse has quality and productivity benefits

Principle of Design with Reuse

Designs that reuse existing assets are better.

- Design with reuse increases design implementability
  - quicker way of building by reusing existing assets

Principle of Design for Reuse

Designs that produce reusable assets are better.

- Reuse is only possible if existing assets are built with reuse in mind
- Could be more complex to reusable assets that single (specific) use assets

Aesthetic Principles

- Beauty of software
  - David Gelernter argues that beauty is a function of simplicity and power
  - Power is the ability to get many tasks done.
  - If a software is simple and at the same time highly flexible (capable of many tasks), it is considered beautiful!