Design and Information Hiding

15-214:
Foundations of Software Engineering

Jonathan Aldrich

Related Reading: D. L. Parnas. *On the Criteria To Be Used in Decomposing Systems into Modules.* CACM 15(12):1053-1058, Dec 1972.

Some ideas from David Notkin's CSE 503 class

What makes one design better than another?

- Not **what** result is produced, or whether it is right
- Instead, ***quality attributes***
  - **How** the result is produced
  - **Characteristics** of the code
- Examples:
  - **Evolvability**– ability to easily add and change capabilities
  - **Local reasoning**– ability to reason about parts separately
  - **Reuse** – avoid duplicating functionality
  - **Robustness** – operates under stress or invalid input
  - **Performance** – yields results at a high rate or with low latency
  - Testability, security, fault-tolerance,

Design Case Study: Key Word In Context (KWIC)

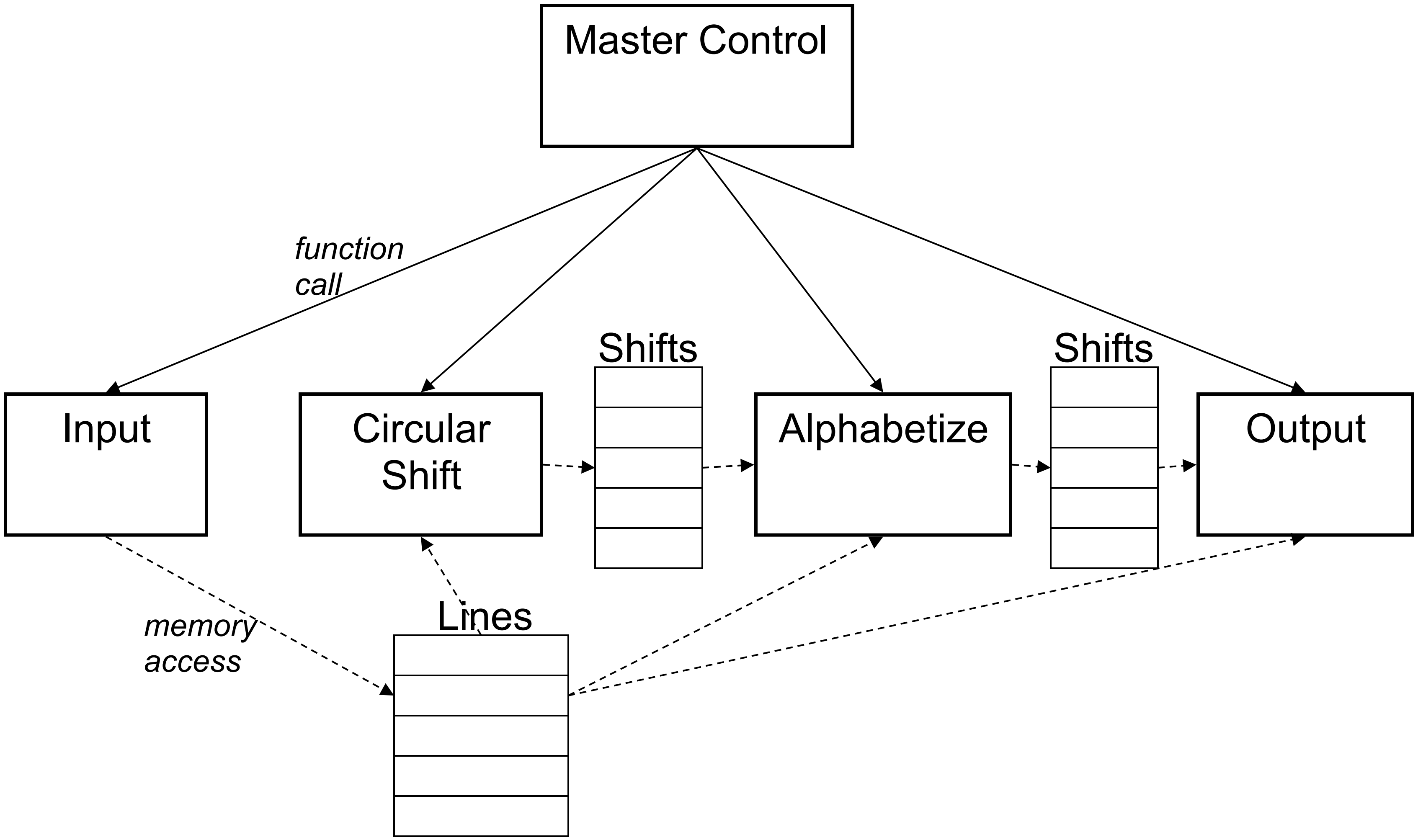
- “The KWIC [Key Word In Context] index system accepts an ordered set of lines, each line is an ordered set of words, and each word is an ordered set of characters. Any line may be "circularly shifted" by repeatedly removing the first word and appending it at the end of the line. The KWIC index system outputs a listing of all circular shifts of all lines in alphabetical order.”
  - Parnas, 1972

- Consider KWIC applied to the title of this slide

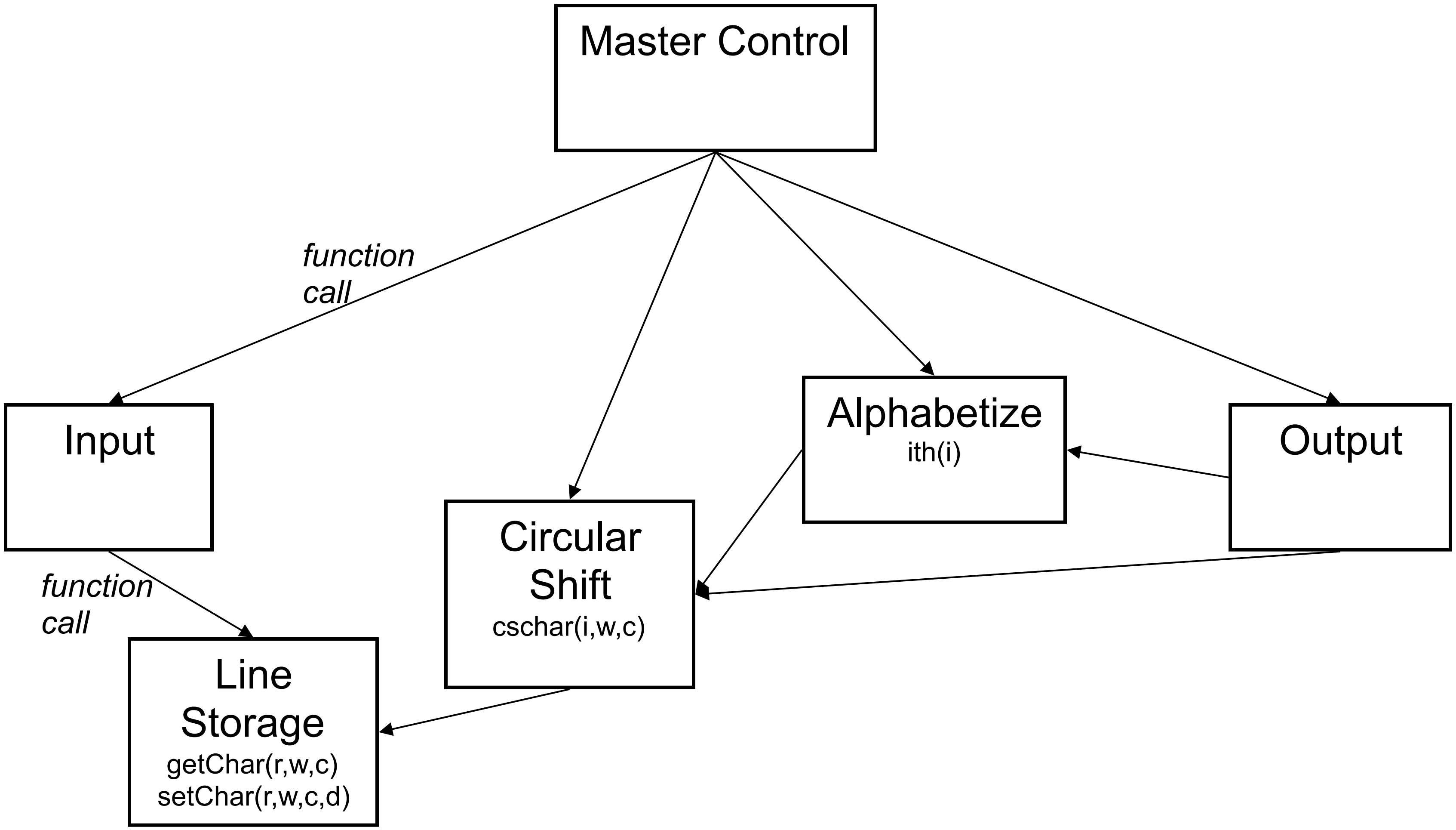
Design Case Study:
Case Study: Design
Study: Design Case
Key Word In Context (KWIC)
Word In Context (KWIC) Key
In Context (KWIC) Key Word
Context (KWIC) Key Word In
(KWIC) Key Word In Context

(KWIC) Key Word In Context
Case Study: Design
Context (KWIC) Key Word In
Design Case Study:
In Context (KWIC) Key Word
Key Word In Context (KWIC)
Study: Design Case
Word In Context (KWIC) Key

KWIC Modularization #1



KWIC Modularization #2



KWIC Observations

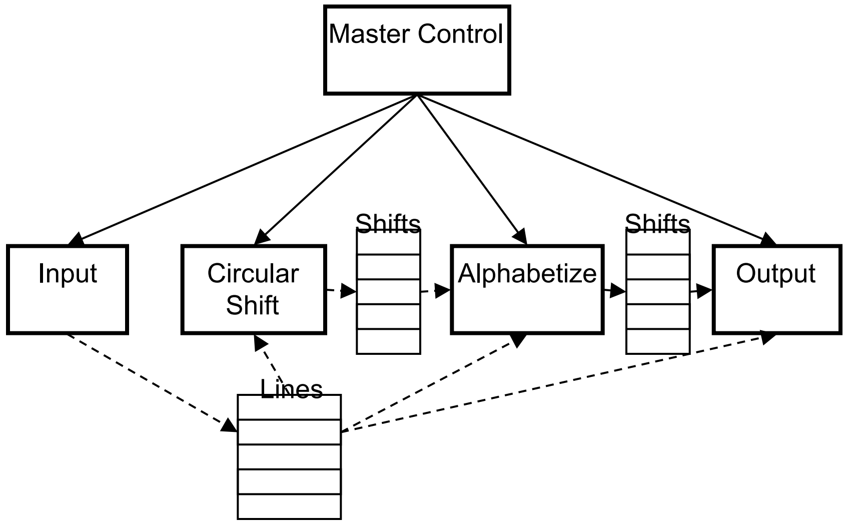
- Similar at run time
  - May have identical data representations, algorithms, even compiled code
- Different in code
  - Understanding
  - Documenting
  - Evolving

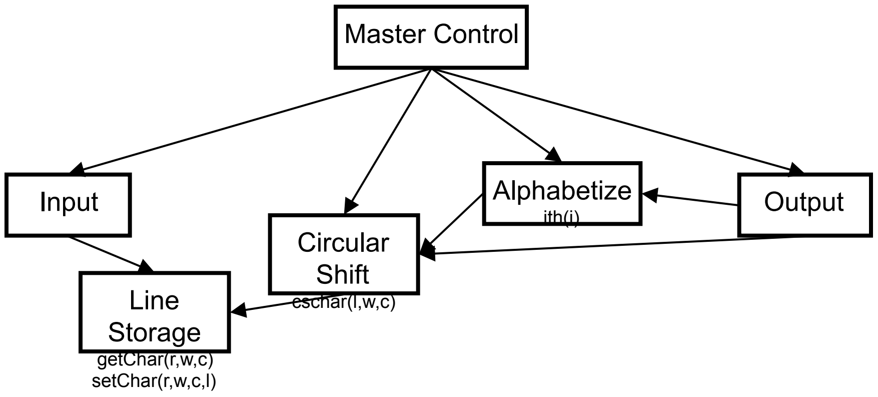
Software Change

- …accept the fact of change as a way of life, rather than an untoward and annoying exception.
  —Brooks, 1974
- Software that does not change becomes useless over time.
  —Belady and Lehman
- For successful software projects, most of the cost is spent evolving the system, not in initial development
  - Therefore, reducing the cost of change is one of the most important principles of software design

Effect of Change?

- Change input format
- Don't store all lines in memory at once
- Use an encoding to save line storage space
- Store the shifts directly instead of indexing
- Amortize alphabetization over searches





Effect of Change?

- Change input format
  - Input module only
- Don’t store all lines in memory at once
  - Design #1: all modules
  - Design #2: Line Storage only
- Use an encoding to save line storage space
  - Design #1: all modules
  - Design #2: Line Storage only
- Store the shifts directly instead of indexing
  - Design #1: Circular Shift, Alphabetizer, Output
  - Design #2: Circular Shift only
- Amortize alphabetization over searches
  - Design #1: Alphabetizer, Output, and maybe Master Control
  - Design #2: Alphabetizer only

Other Factors

- Independent Development
  - Data formats (#1) more complex than data access interfaces (#2)
  - Easier to agree on interfaces in #2 because they are more abstract
  - More work is independent, less is shared
- Comprehensibility
  - Design of data formats in #1 depends on details of each module
  - More difficult to understand each module in isolation for #1

A Note on Performance

- Parnas says that if we are not careful, decomposition #2 will run slower
- He points out that a compiler can replace the function calls with inlined, efficient operations
- Lesson: don't prematurely optimize
  - Smart compilers enable smart designs
  - Evolvability usually trumps the overhead of a function call anyway

Decomposition Criteria

- Functional decomposition
  - Break down by major processing steps
- ***Information hiding*** decomposition
  - Each module is characterized by a design decision it hides from others
  - Interfaces chosen to reveal as little as possible about this

Information Hiding

Derived from definition by Edward Berard – concept due to Parnas

- Decide what design decisions are likely to change and which are likely to be stable
- Put each design decision likely to change into a module
- Assign each module an interface that hides the decision likely to change, and exposes only stable design decisions
- Ensure that the clients of a module depend only on the stable interface, not the implementation

- Benefit: if you correctly predict what may change, and hide information properly, then each change will only affect one module
  - That's a big if…do you believe it?

Hiding design decisions

Information hiding is ***NOT*** *just about data representation*

Decision	Mechanism
• Data representation	
• Platform	
• I/O format	
• User Interface	
• Algorithm	

Hiding design decisions

- Algorithms – procedure
- Data representation – abstract data type
- Platform – virtual machine, hardware abstraction layer
- Input/output data format – I/O library
- User interface – model-view pattern

What is an Interface?

- Function signatures?
- Performance?
- Ordering of function calls?
- Resource use?
- Locking policies?

- Conceptually, an interface is everything clients are allowed to depend on
  - May not be expressible in your favorite programming language

Correspondence

- How well the design of the code matches the requirements
- If each requirement is implemented by a separate module, then a change in a requirement should only require changes to one module
  - Hard to achieve in practice
  - OO approaches design code after a model of the world
    - This helps, but some requirements crosscut the structure of the world as well!
- Separation of Concerns
  - Generalizes correspondence to “concerns” that may be implementation issues, not just requirements

Design Practices

Specification: The Starting Point for Design

- Functionality
  - Usually a set of use cases
    - Detailed scenarios of system use
    - Includes normal and exceptional cases
  - Less often: mathematical specifications

- Quality attributes
  - **Expected areas of extension**
  - Robustness, Security
  - Performance, Fault-tolerance

- We'll talk more about specifications and requirements gathering later

Example: Use Cases

Example: Quality Attributes

Identifying Classes: Noun Extraction

- Start with short problem description
- Identify the nouns and analyze
  - External entities: leave out
    - unless system needs to model them
    - example: “The User”
  - Tangible entities: classes
  - Abstract nouns: classes or attributes (fields)
    - weight, brightness, size
    - Complex abstract nouns might end up as a class
      - e.g. Color, Message, Event
- Add
  - Boundary classes: interaction with world
    - Typically one per screen/dialog
  - Control classes: encapsulate non-trivial computations
  - Data structures that support the entities
  - Classes for abstract implementation concepts
    - Controller, Router, Manager, …

What should be a Class?

- Retained information
  - Need to remember data about the object
- Needed services
  - Operations that change attribute values or compute information
- Multiple attributes
  - Class groups data related by a concept
  - No class usually needed for a scalar
- Common attributes & operations
  - A set of attributes/operations is common to many objects
- Essential requirements
  - Entities in the problem space

Source: [Coad and Yourdon 91]

Example: Noun Extraction

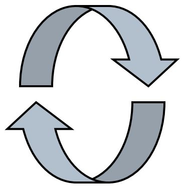
Abstract Design: CRC Cards

- Class-Responsibility-Collaboration
  - Name of class
  - Responsibilities/functionality of the class
  - Other classes it invokes to achieve that functionality

- Responsibility guidelines
  - Spread out functionality
    - No “god” classes – make maintenance difficult
  - State responsibilities generally
    - More reusable, more abstract
  - Group behavior with related information
    - Enhances cohesion, reduces coupling
    - Promotes information hiding of data structures
  - Information about one thing goes in one place
    - Spreading it out makes it hard to track

CRC Validation

- Validation
  - Ensure all functionality in specification is covered by some class
  - Reason through how functionality could be achieved
    - Abstractly executing the program
    - What other classes are needed?
    - Are their responsibilities enough for this class to do what it needs to do?
- Refine as needed



Example: CRC Cards

Attributes, Associations, and Operations

- Go through use cases
  - Attribute: something that belongs to a class
    - Needed for computation in the use case
  - Association: one class stores another
    - Usually implemented by a field or collection—but keep abstract early in design
  - Operation: verbs in use cases
    - OO: usually goes in the object on which the verb operates
    - Categories
      - accessors: access data
      - mutators: manipulate data
      - computational methods

Quality Attributes

- So far, we've focused on capturing functionality in a design

- But *good* design is primarily about *quality attributes*, e.g.
  - *Extensibility* – ability to easily add and change capabilities
  - *Robustness* – operate under stress or invalid input
  - *Usability* – ability for users to easily accomplish tasks
  - *Security* – withstand attacks
  - *Fault-tolerance* – recover from component failure
  - *Performance* – yields results at a high rate or with low latency

Refining a Design

- Step through Use Cases
  - Verify completeness of diagram by asking:
    - Which methods execute?
    - What methods are called?
    - What does each method or object have to know?

- Consider quality attributes
  - Make concrete with a test
    - e.g. modification scenario, performance target
  - Generate multiple designs – ***NOT JUST ONE!***
    - What design patterns achieve this attribute?
    - May be helpful to have different people develop designs independently
  - Evaluate designs
    - How well does this design achieve the entire set of quality attributes?
    - May require prioritizing attributes

Cohesion

- The number of dependences within a module
- High cohesion is good
    - Changes are likely to be local to a module
    - Easier to understand a module in isolation

Coupling

- The number of dependences between modules
  - may be syntactic or semantic
- Costs of high coupling
  - change to an interface affects other modules
  - difficulty understanding or reusing code
- Coupling increases over time
  - I need to use that function over there…

Coupling and Information Hiding

- Coupling
  - How many dependencies?
  - Proxy for cost of interface change
- Information hiding
  - Depend only on stable design decisions
  - Incorporates likelihood of interface change
    - Thus a more direct measurement of a design's value
- Sometimes coupling is OK!
  - High coupling between framework and client
  - Framework interface captures assumptions that don't change between applications
    - Also hides framework implementation decisions that are likely to change
  - Client encapsulates code specific to an application