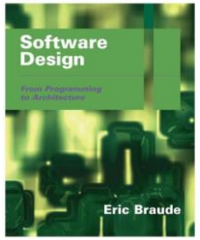
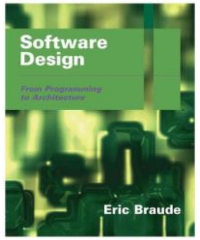


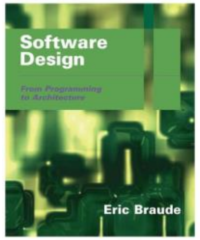
Lec 06 - Design Principles I

Topic covered



- Introducing two design Principles:
  - Correctness
  - Robustness

Correctness

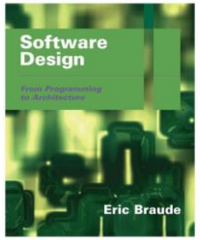


KEY CONCEPT

Correctness

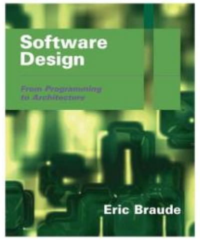
Goal: That each artifact satisfies designated requirements, and that together they satisfy all of the application’s requirements.

Approaches to correctness



- ✧ How can we know that a design is correct or even sufficient?
- ✧ Approaches to correctness.
  - Informal approaches.
  - Formal approaches .
- ✧ Informal approaches: to be convinced that the design covers the required functionality.
- ✧ Formal approaches
  - Formal methods for establishing correctness involve applying mathematical logic to analyzing the way in which the variables change.
  - Formal methods are usually applied when the design enters the detailed design.

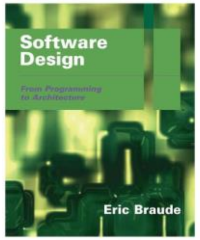
Informal approaches to correctness



- Correctness by Informal Methods
  Simplify and modularize designs until they are convincing.

- Informal Approaches
  - Informal approaches are based on the common sense idea that before we can proclaimed a design to be correct, we have to understand it completely.
  - Informal approaches require that design must be
    - Readable (to enhance understanding)
    - Modular   (to deal with complexity)

Sufficient Designs: Terminology and Rationale



Minimum goal:

A design sufficient to implement the requirements.

Sometimes called …

a correct design

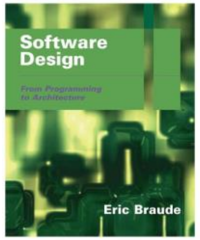
It follows that …

the design must be entirely understandable

A common way to achieve this is to make …

the design very modular

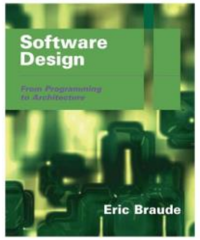
Formal approaches to correctness



✧ Formal approaches

- Keeping variable changes under tight control.
- It can be achieved through invariants.
- Invariants are unchanging relationships among variable values.
- Invariants used at class level are *class invariants.*
- Examples:
    - “length>=0”
    - “length * breadth == area”

Invariants for Class Automobile



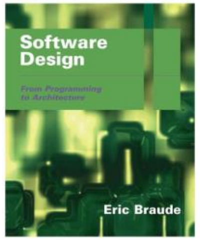
-- with variables *mileage*, *VehicleID*, *value*, *originalPrice*, and *type*:

1) *mileage* > 0
2) *mileage* < 1000000
3) *vehicleID* has at least 8 characters
4) *value* >= -300

- ($300 is the disposal cost of a worthless automobile)

5) *originalPrice* >= 0
6) (*type* == "REGULAR" && *value* <= *originalPrice*) ||

- (*type* == "VINTAGE" && *value* >= *originalPrice*)

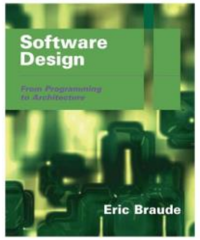


Formal approaches to correctness (cont.)

✧ Some guidelines for achieving correctness at coding level.

- Make variables private.
- Change the variables values only through public accessor methods.
- Accessors can be coded to maintain invariants.

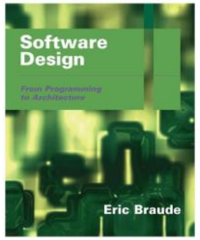
Interfaces to modules



- ✧ Interfaces: collections of function prototypes: Make designs more understandable.

- ✧ Modularity
  - Modularization is key to assess the correctness of a design
  - A module can be either a class or a package of classes
  - An interface is a set of functions forms (or prototypes).

Interfaces to modules



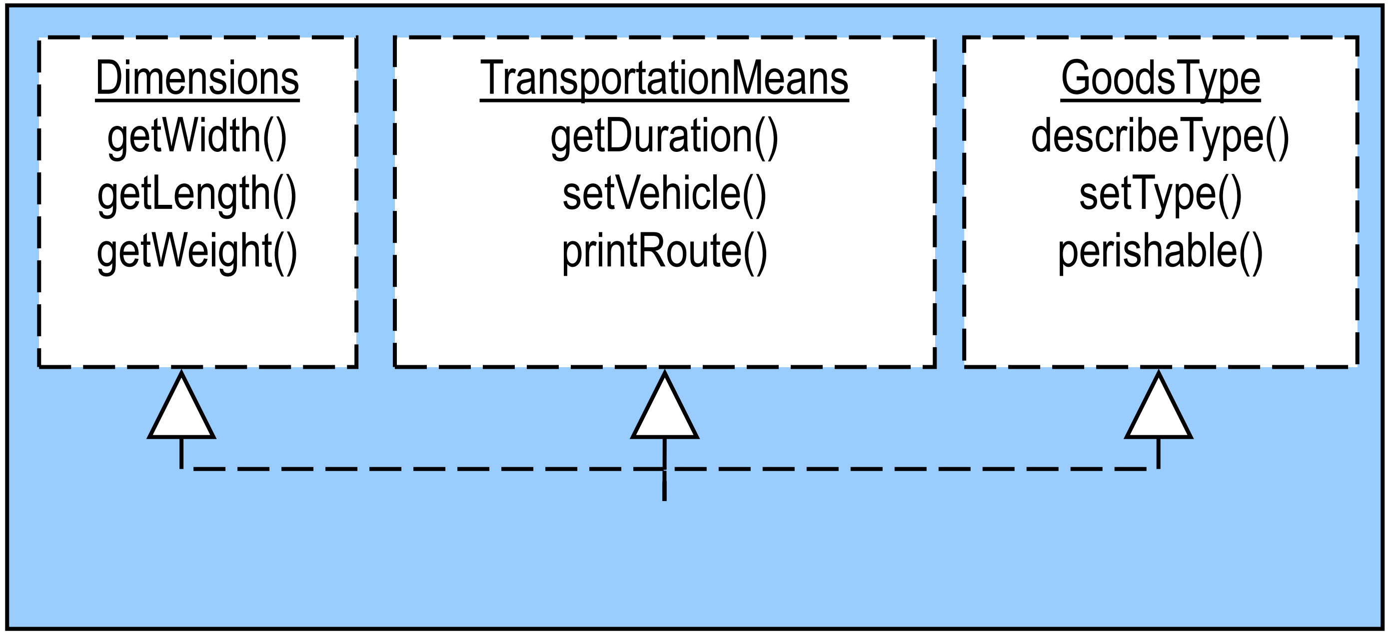
✧ Interfaces to classes

- When a class supports many methods, it is often beneficial to group them into several interfaces

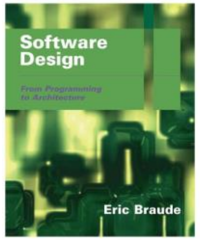
- Grouping allows reuse

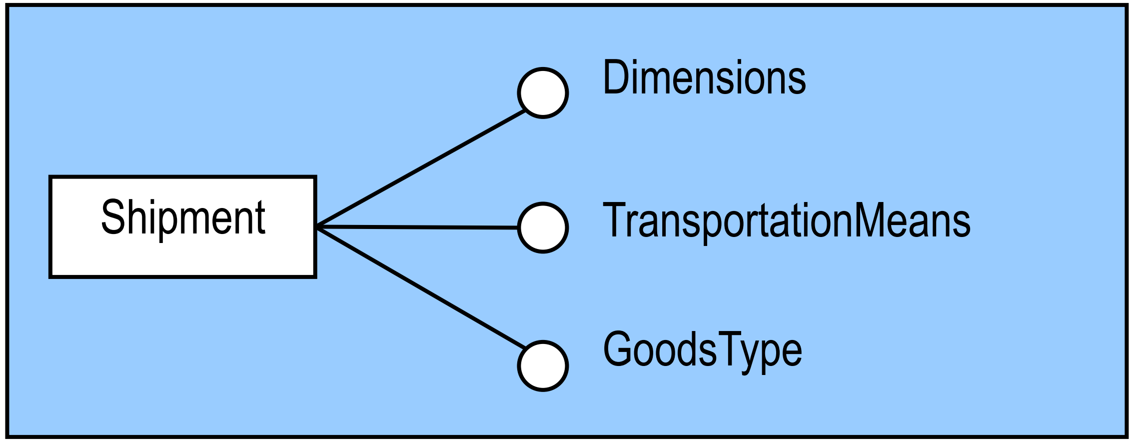
```
Shipment
setVehicle()
perishable()
getWidth()
printRoute()
describeType()
getLength()
getDuration()
setType()
```

Introducing Interfaces



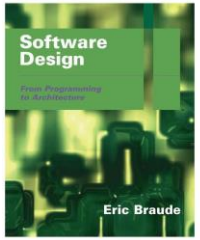
Introducing Interfaces






```
Shipment
setVehicle()
perishable()
getWidth()
printRoute()
describeType()
getLength()
getDuration()
setType()
```

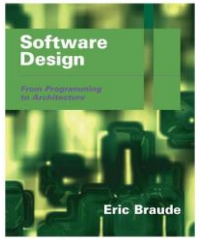
Interfaces to modules

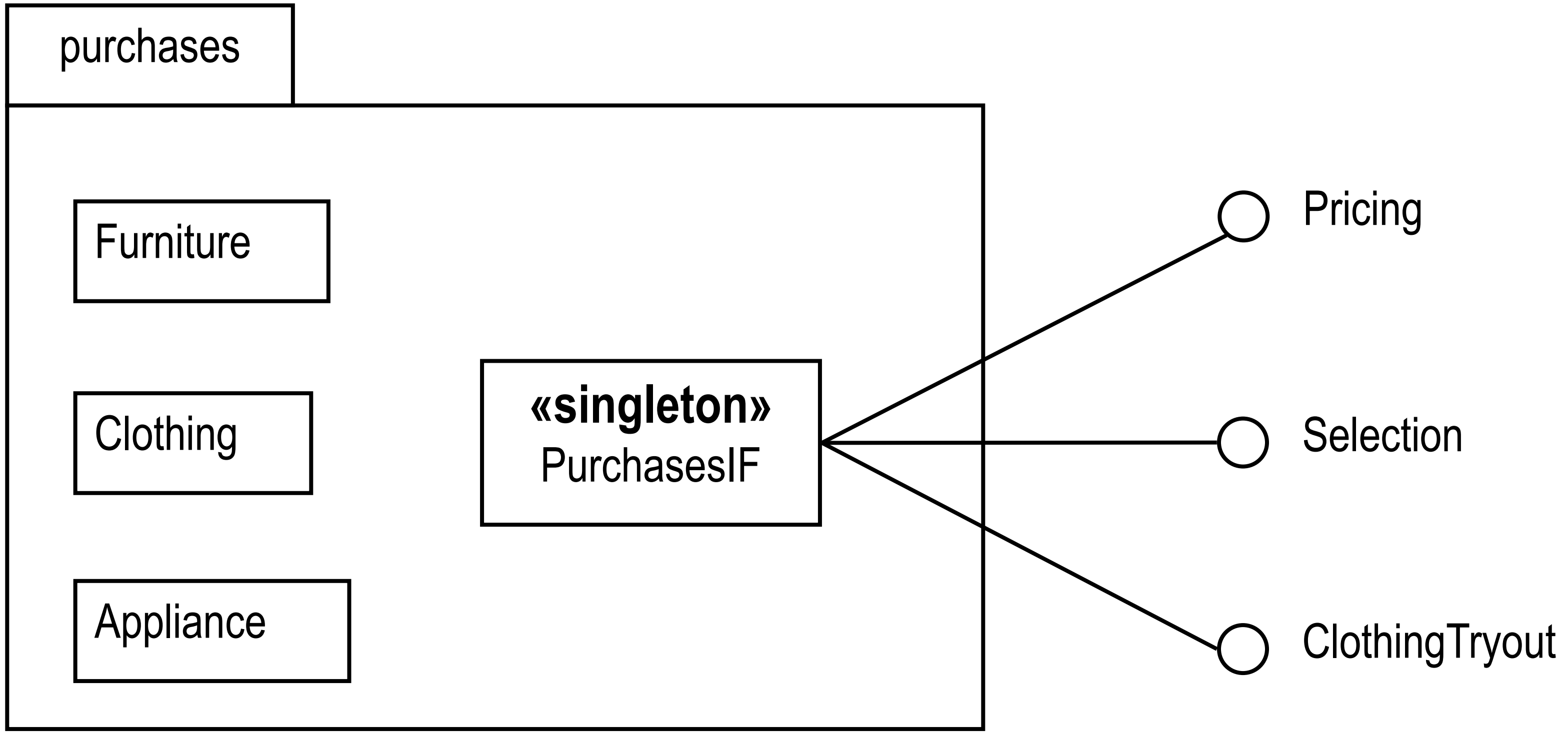


✧ Interfaces to Packages

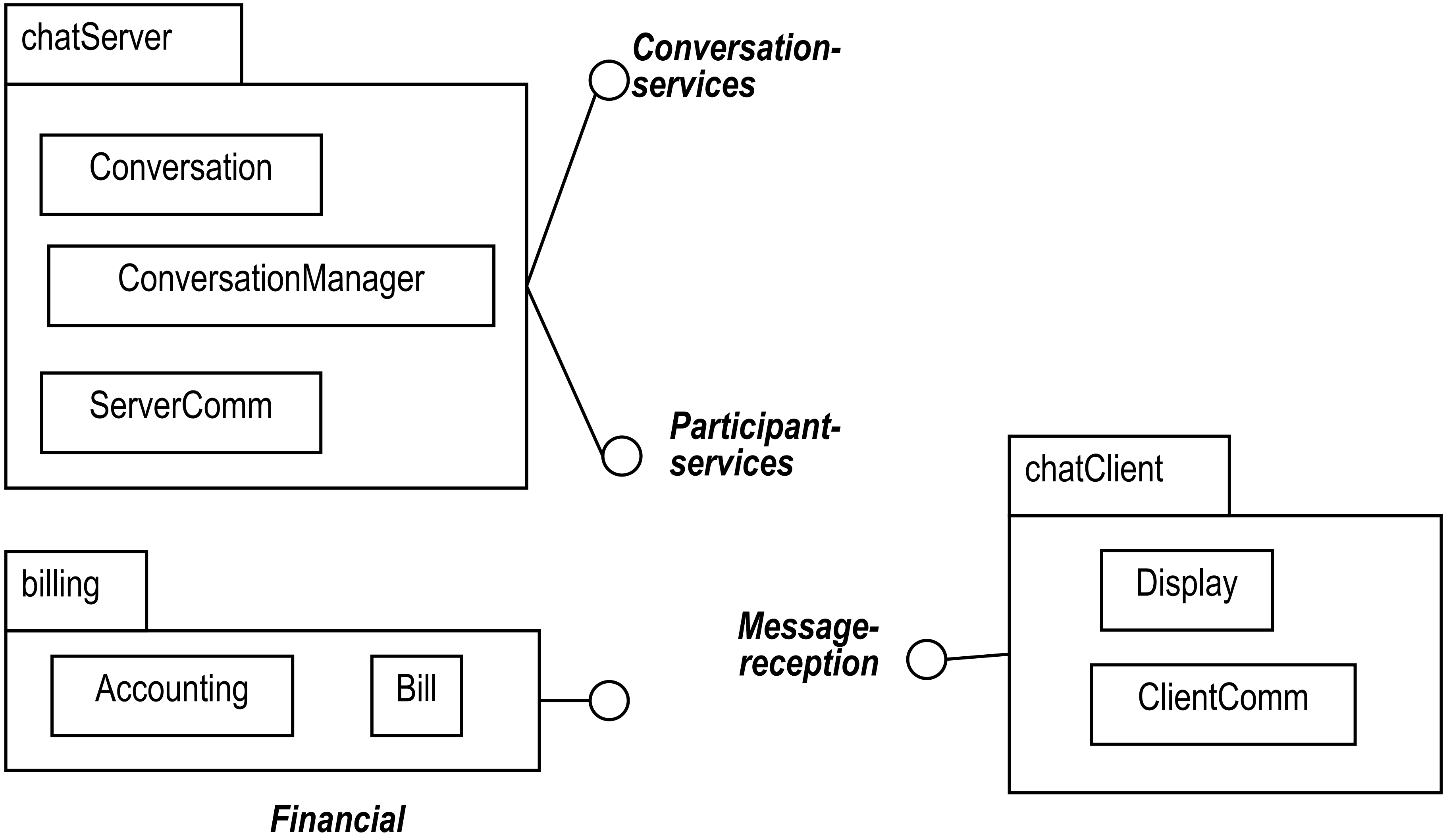
- Interface to package is different idea than an interface to a class
- Provide an interface for a designated object of a class in the package or
- Define a class in a package and define its interface as static methods

Package Interfaces

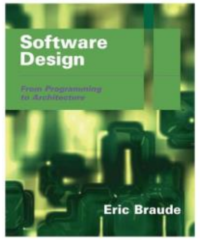




Example of Package Interfaces



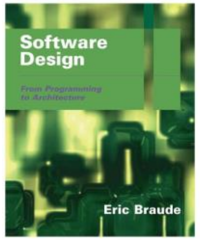
Modularization



- ✧ To modularize an object-oriented application
  - Create packages at the higher level
  - Create classes at the lower level

- ✧ Choosing classes (Two kinds of classes)
  - Domain classes
    - classes that pertain to the specific application under design.
    - Can be obtained from the sequence diagrams of use cases.
  - Non-domain classes
    - generalization of domain classes

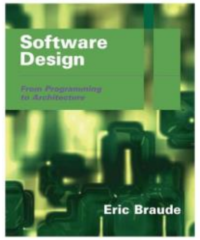
Domain vs. Non-Domain Classes



- *Domain classes*: Particular to the application
  - Examples: *BankCustomer, BankTransaction*, *Teller*
  - Typically not GUI classes
  - Sufficient to classify all requirements

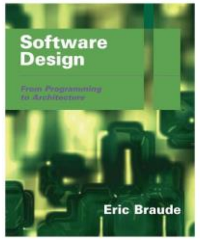
- *Non-Domain classes*: Generic
  - Examples: abstract classes, utility classes (e.g. *Person: teller or customer*)
  - Arise from design and implementation considerations

Robustness

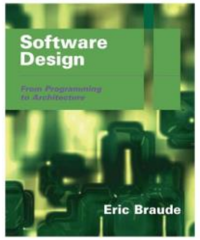


- ✧ The ability to handle anomalous situations
  - Incorrect user input
  - Faulty data communication
  - Developer errors
- ✧ Verifying Input.
  - Robustness is promoted by verifying data values before using them.
  - Check all inputs for constraints. It can include:
    - Type verification, Preconditions, Invariants, Postconditions
- ✧ Initialize variables and objects at declaration/ creation time improve robustness

Sources of Errors



- ✧ Robustness --- ability to handle anomalous situations even in the presence of errors
- ✧ Sources of error:
  - Faulty input
    - User input
    - Input, not from users
      - Data communication
      - Function calls made by other applications
  - Developer errors
    - Faulty design
    - Faulty implementation



Summary

- ✧ Correctness of a Design or Code
  - Supports the requirements
  - In general, many correct designs exist

- ✧ Robustness of a Design or Code
  - Absorbs errors
    - -- of the user
    - -- of developers