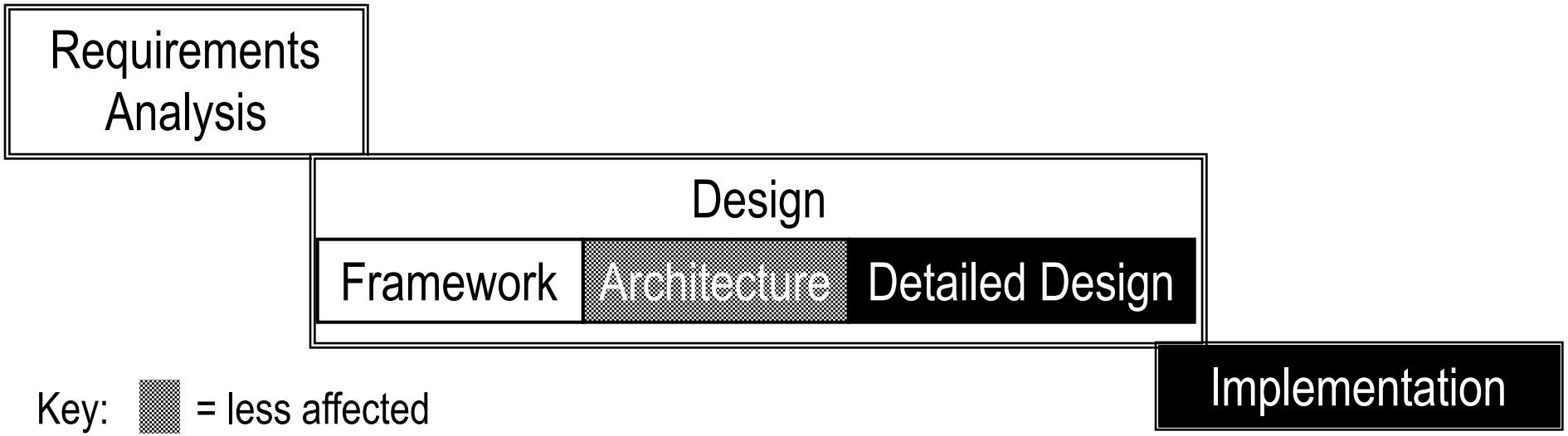
Chapter 4
Design Principles I
Correctness and Robustness

Process Phase Affected by This Chapter



Key Concept: → Correctness ←

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

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

Key Concept:
→ Correctness by Informal Methods ←

Simplify and modularize designs until they convince.

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

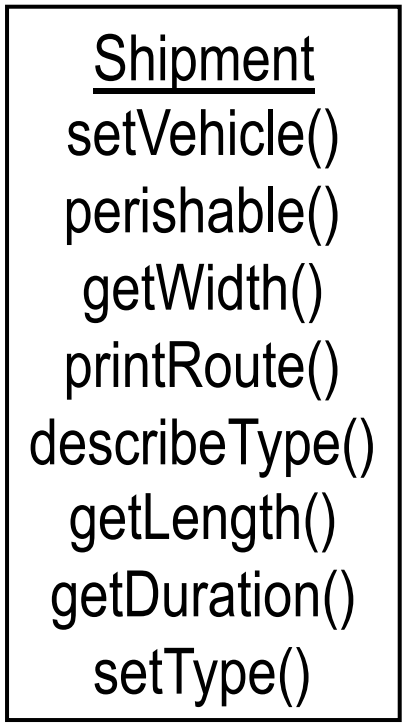
($300 is the disposal cost of a worthless automobile)

5) *originalPrice* >= 0

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

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

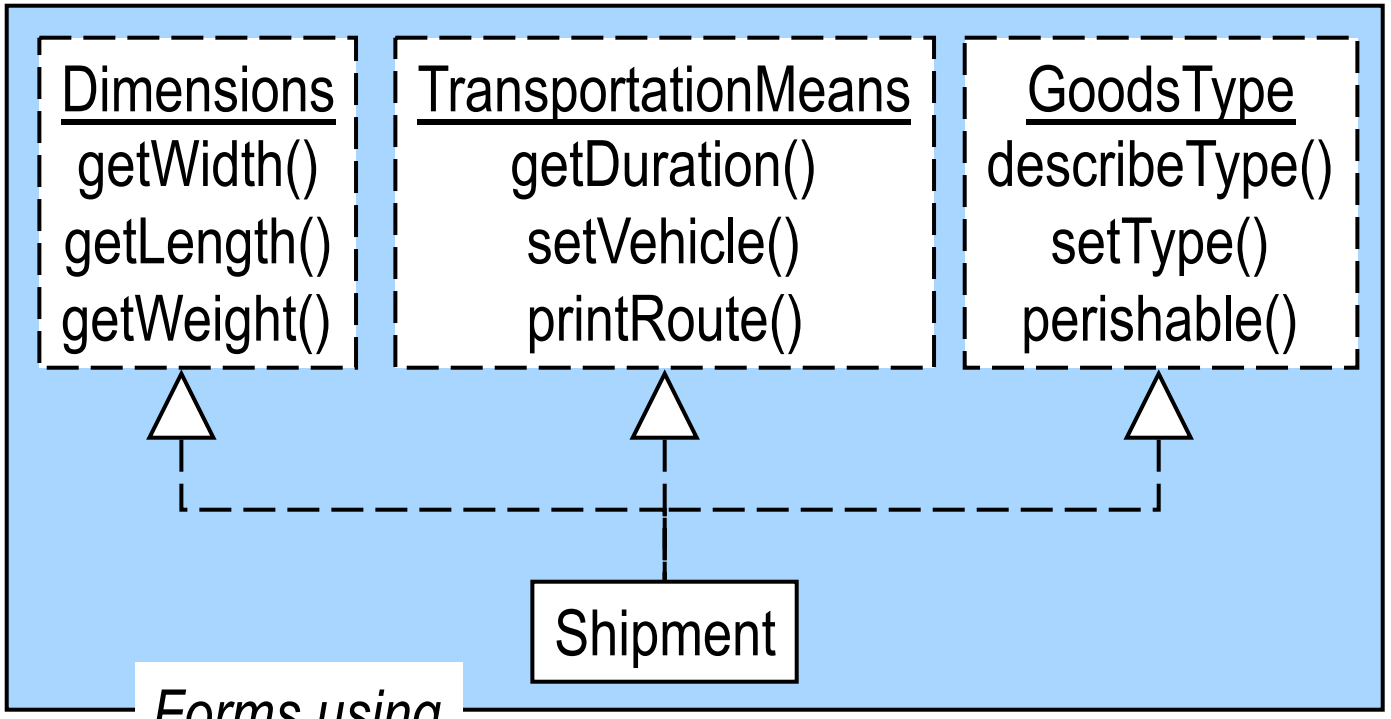
Introducing Interfaces *1 of 2*



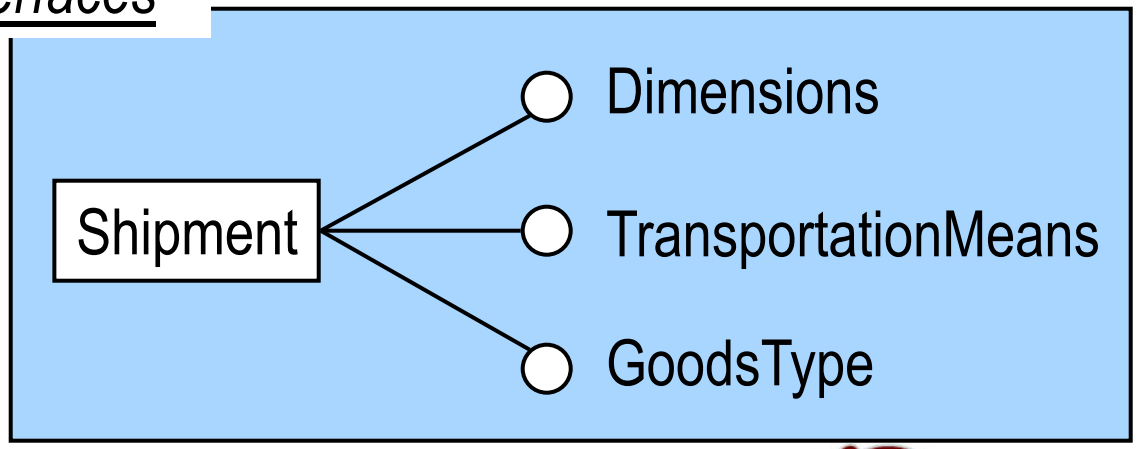
Introducing Interfaces *2 of 2*

Original form

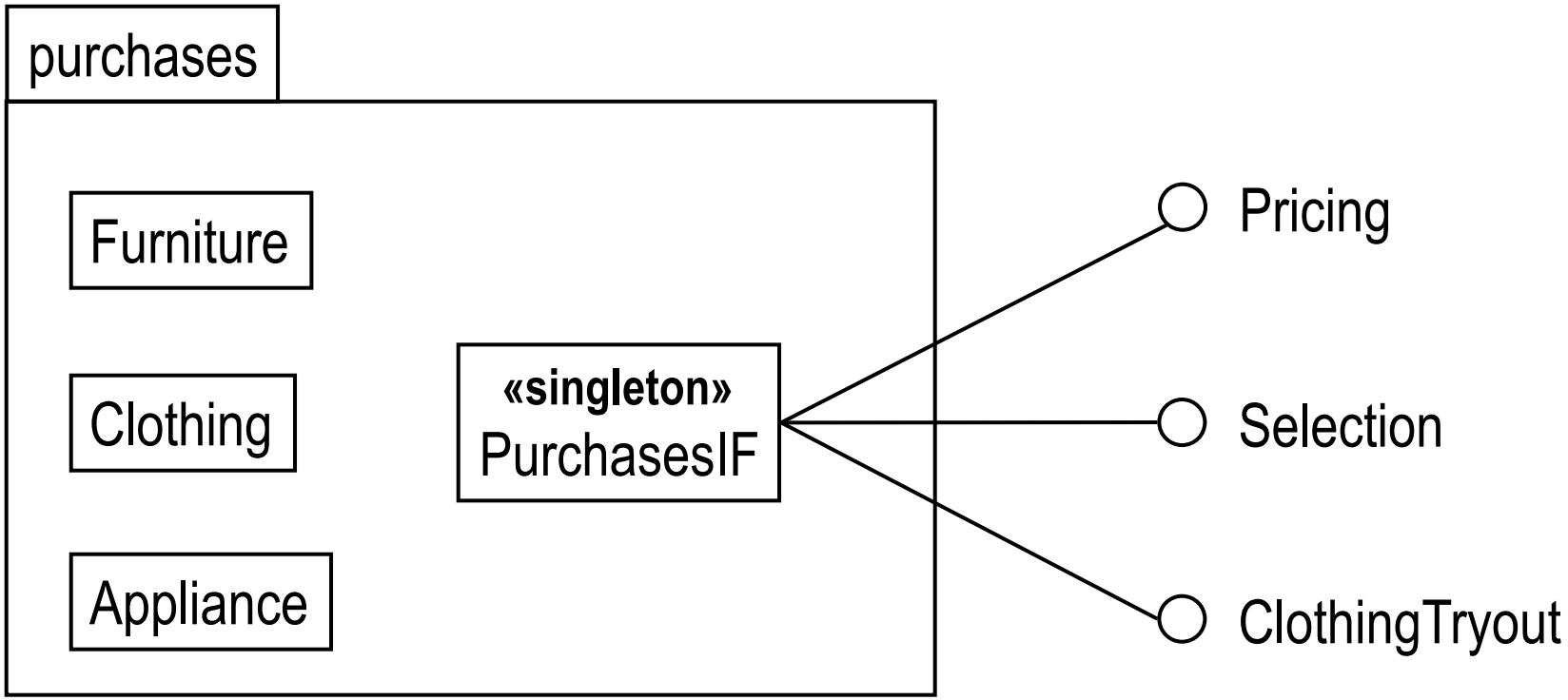
Shipment
setVehicle()
perishable()
getWidth()
printRoute()
describeType()
getLength()
getDuration()
setType()



Forms using interfaces

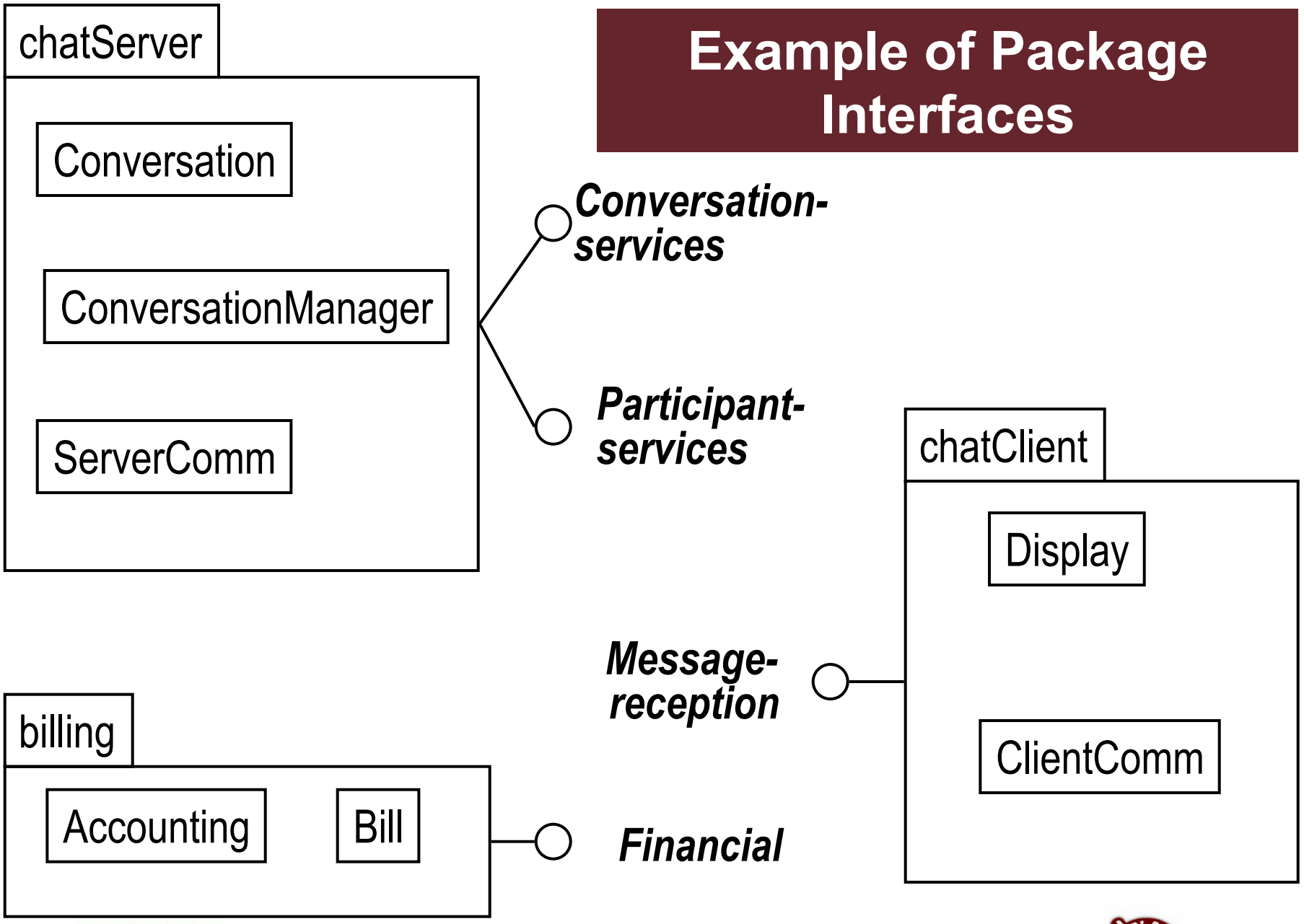


Package Interfaces



Adapted from *Software Design: From Programming to Architecture* by Eric J. Braude (Wiley 2003), with permission.

Example of Package Interfaces



Adapted from *Software Design: From Programming to Architecture* by Eric J. Braude (Wiley 2003), with permission.

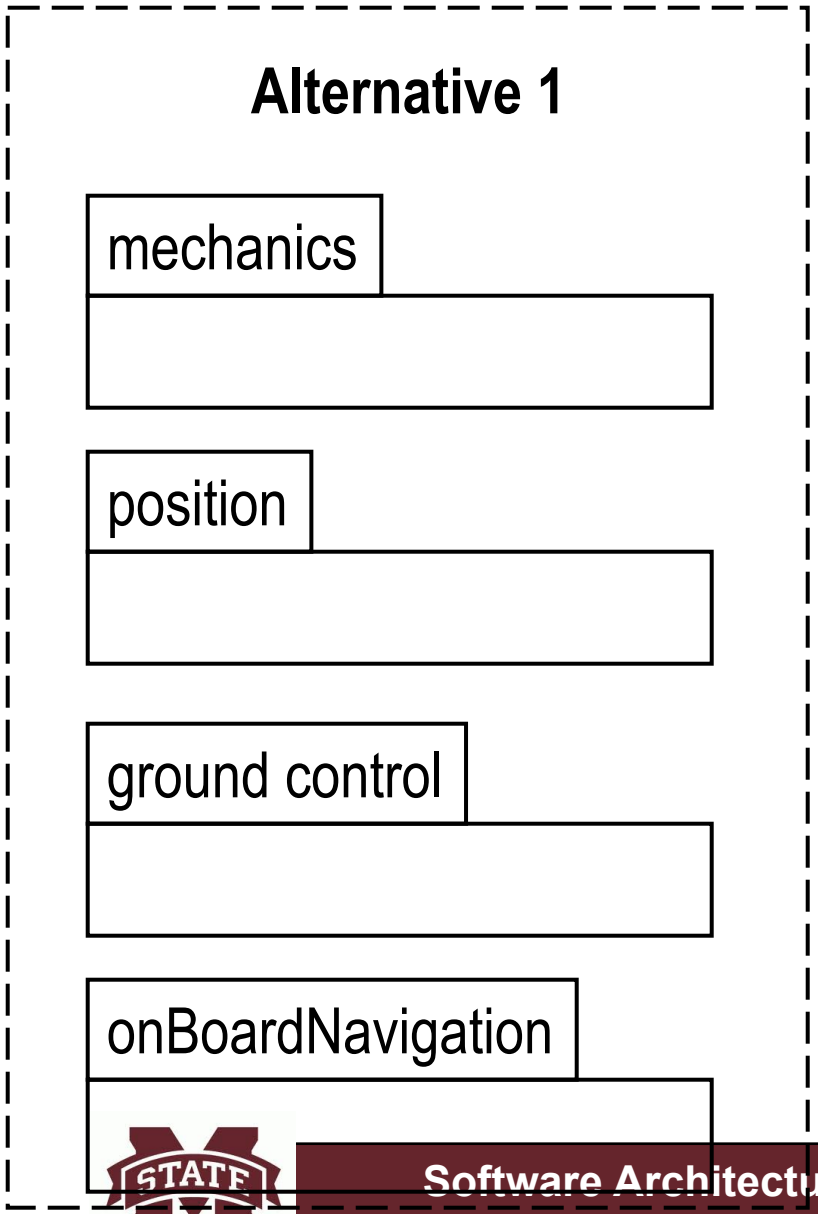
Key Concept: → Interfaces ←

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

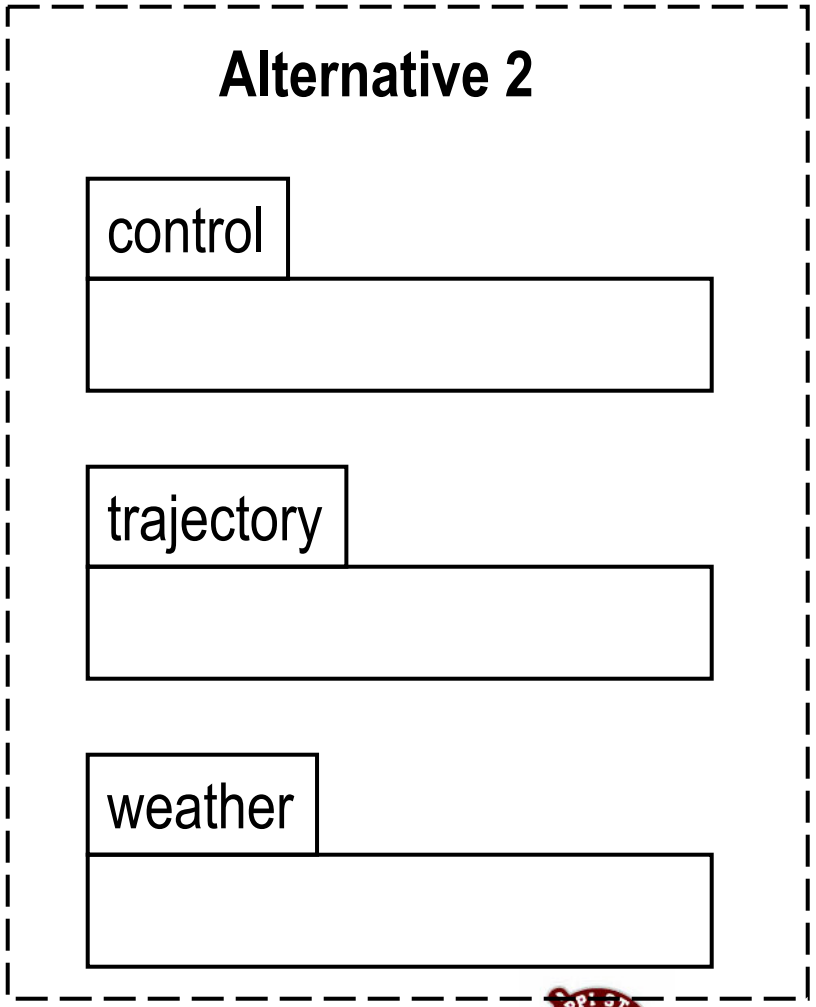
Domain vs. Non-Domain Classes

- ***Domain classes*: Particular to the application**
  - **Examples: *BankCustomer, BankTransaction*, *Teller***
  - **Typically not GUI classes**
  - **Sufficient to classify all requirements (see chapter xx)**
- ***Non-Domain classes*: Generic**
  - **Examples: abstract classes, utility classes**
  - **Arise from design and implementation considerations**

Alternative Modularizations



Application tracking trajectory of rocket carrying orbit-bound satellite into position



Adapted from *Software Design: From Programming to Architecture* by Eric J. Braude (Wiley 2003), with permission.

Improving Robustness: Sources of Errors

Protection from faulty Input

- User input
- Input, not from user
  - Data communication
  - Function calls made by other applications

Protection from developer error

- Faulty design
- Faulty implementation

Constraints on Parameters

Example:

int computeArea(int aLength, int aBreadth) { … }

- ***Capture parameter constraints in classes if feasible***

  int computeArea(RectangleDimension a RectangleDimension)

- ***Specify all parameter constraints in method comments***

  aLength > 0 and
  aBreadth > 0 and
  aLength >= *aBreadth*

- ***Callers obey explicit requirements on parameters***
  - Problem is method programmers have no control over callers
- ***Check constraints first within the method code***

  if(aLength <= 0) ……
  - Throw exception if this is a predictable occurrence
  - Otherwise abort if possible
  - Otherwise return default if it makes sense in context
    - And generate warning or log to a file

Key Concept: → Robustness ←

-- is promoted by verifying data values before using them.

Wrapping Parameters

Replace int computeArea(int aLength, int aBreadth)

{..}

with int computeArea(Rectangle aRectangle)

{..}

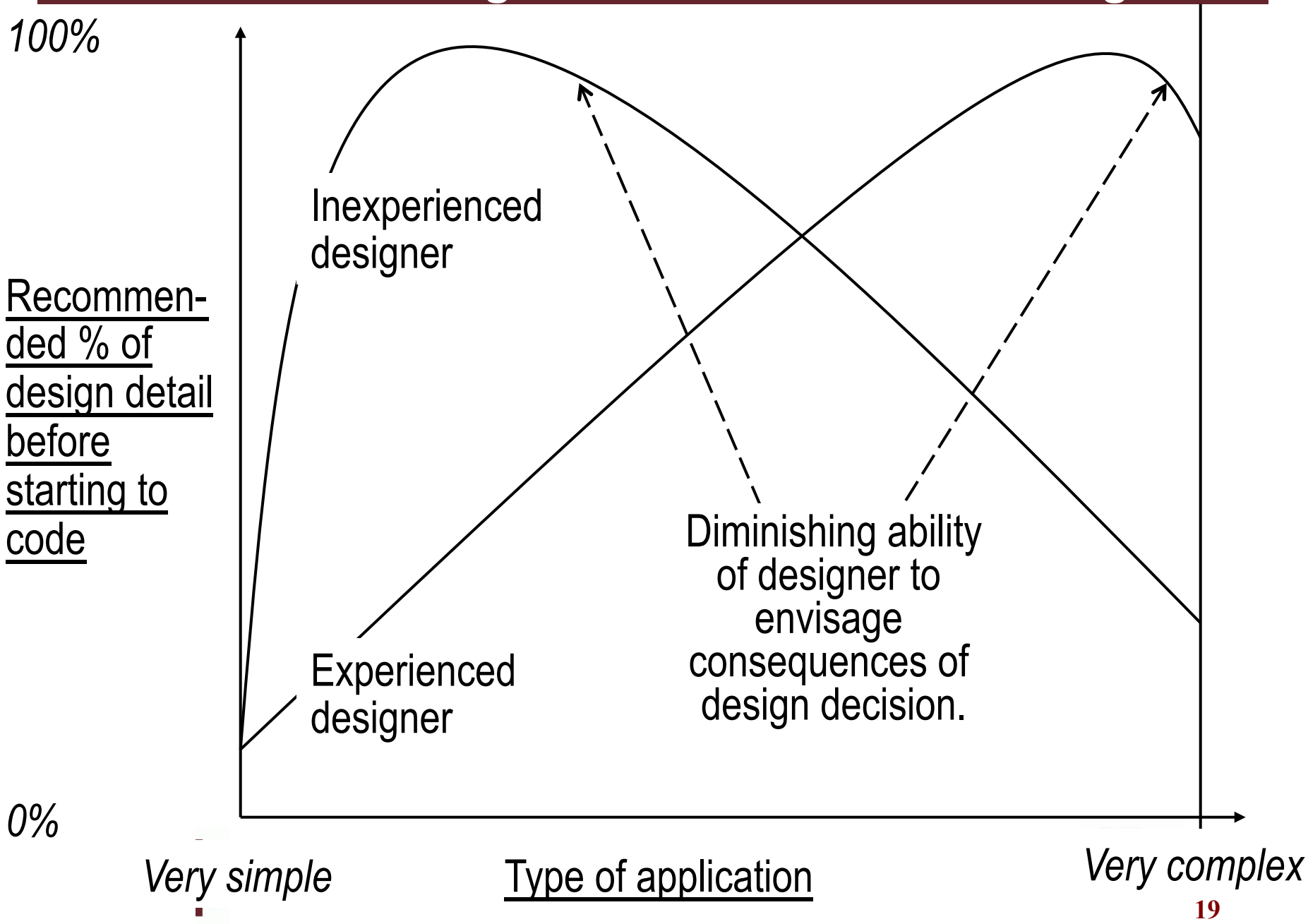
-- where class Rectangle

```
{ …
    Rectangle( int aLength, int aBreadth )
        {   if( aLength > 0 ) this.length = aLength;
            else …..
         }
… }
```

Key Concept: → Robustness ←

-- is promoted by enforcing intentions.

How Much Design Detail Before Initial Coding?



Video Store Application: Sufficient Classes?

Video

CheckOutDurationDisplay

Customer

CheckOutDisplay

BarCodeReader

RegisterNewVideoDisplay

Summary of This Chapter

- ***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**