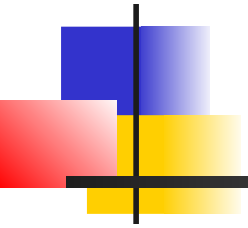


Designing Classes

Identifying Classes

During design, you need to identify the classes.

Classes are used to **model** things:

- Tangible things in the design: *nouns*, such as
  - purse, mailbox, bank account, keypad, email message
- Users
  - Student, Customer, Cashier
- Agents ("-er" or "-or" words)
  - Scanner, file reader, sentence reader, CSVreader
  - agents perform some operation (scanner reads strings, etc.)
- Events and Transactions
  - Button click, ATM Withdrawal, Register for Course
  - transactions are uses for interactions between classes, usually contain info that doesn't "belong" in a *thing* class

Identifying Classes (2)

- Interfaces and Devices
  - connection to a database -- "a connection"
  - Printer, File, URL
- Foundation Classes
  - String,
  - List, Queue,
  - Rectangle,
  - Button
  - "data types" with well-known properties: Date, Money

Identify Responsibilities

- *Verbs* in specification often are responsibilities

```
deposit

withdraw

getBalance

setName

isFull?
```

Assigning Responsibilities

A class should have only one purpose

(all its responsibilities are related to the purpose)

- Purse: responsible for managing coins in a purse

- More than one CLOSELY related responsibility OK
  - Bank: purpose is to manage bank accounts and assets

- A class should have well defined responsibilities
  - ATM: communicate with the client and execute transactions (but not verify them... that's the Bank's job)

CRC cards to Assign Responsibilities

CRC card approach:

no more than you can write on a 3x5-inch note card

Product Catalog	
<<responsibilities>>	<<collaborators>>
• know about products for sale in the Store • find a product by name or product ID	Product

How Much Responsibility?

Avoid Bloated Classes

- If a class gets too large, look for ways to *delegate* some responsibility to other classes.
- Put unrelated responsibilities in different classes

Identifying Behavior

- What behavior (methods) must a class have to perform its responsibilities?

A Purse is responsible for managing coins in a purse.

Behavior a purse should have:

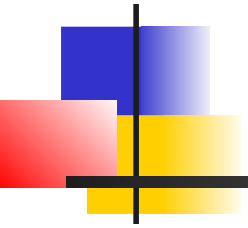
1. insert coins
2. withdraw coins
3. get the balance
4. is it full? purse has a limited capacity

Behavior

- Behavior should be related (*coherent*)
  - don't put unrelated responsibilities into the same class
  - avoid assigning too many behaviors to a class

A Purse is not responsible for:

- printing the balance
- asking the user what he wants
- computing the price of stuff



Designing a Class's Interface

5 Criteria for Good Interface Design

Convenience - has methods that make it easy for programming to access the desired behavior

Clarity - purpose of interface should be clear to the programmer.

Consistency - method names and signatures are consistent with each other, and consistent with those of other, similar classes

Completeness - class provides methods for all required behavior

Cohesion - a class's responsibilities (methods) are all related to a common concept or purpose

What about *Coupling?*

- Coupling is important, too.
- But, coupling is a property of a class's *implementation* not its *interface*.

[Low] Coupling - class doesn't depend to too many other classes.

Exception: coupling to *stable* classes -- such as the Java foundation classes -- is OK.

Interface Design and Class Design

These criteria are related to the class's *interface*:

- Convenience
- Clarity
- Consistency
- Completeness
- Cohesion

This concerned with a class's *implementation:*

- Coupling

Convenience and Clarity

Here are two ways of doing the same thing:

```
if ( list.isEmpty( ) ) break;
if ( list.size() == 0 ) break;
```

which one makes it easier to understand the *purpose* ?

This is an example of **clarity** and **convenience**
(I'm *too lazy* to type "== 0").

Clarity

Get the last element from a list

```
Object e = list.get( list.size() - 1 );
```

// or

```
Object e = list.getLast( );
```

Get the first element from a list

```
Object f = list.get( 0 );
```

// or

```
Object f = list.getFirst( );
```

Completeness

BinaryTree

BinaryTree( )

add(E obj) : boolean

size( ) : int

height( ) : int

isEmpty( ) : boolean

Is this good enough for a BinaryTree?

What methods are missing?

delete(E obj)_____

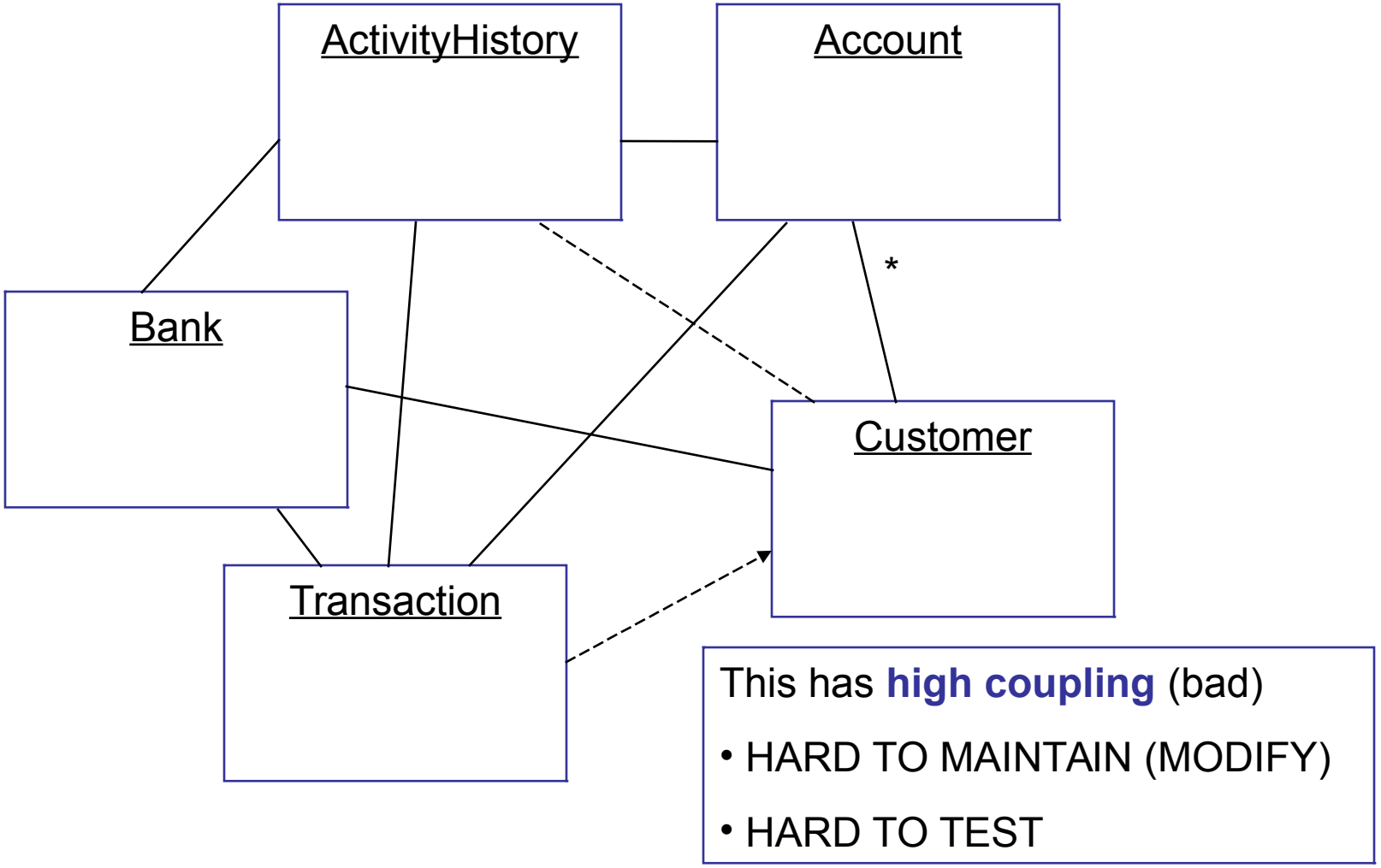
contains(E obj): boolean

_____

_____

Fails to meet the design criterion of _____

Coupling

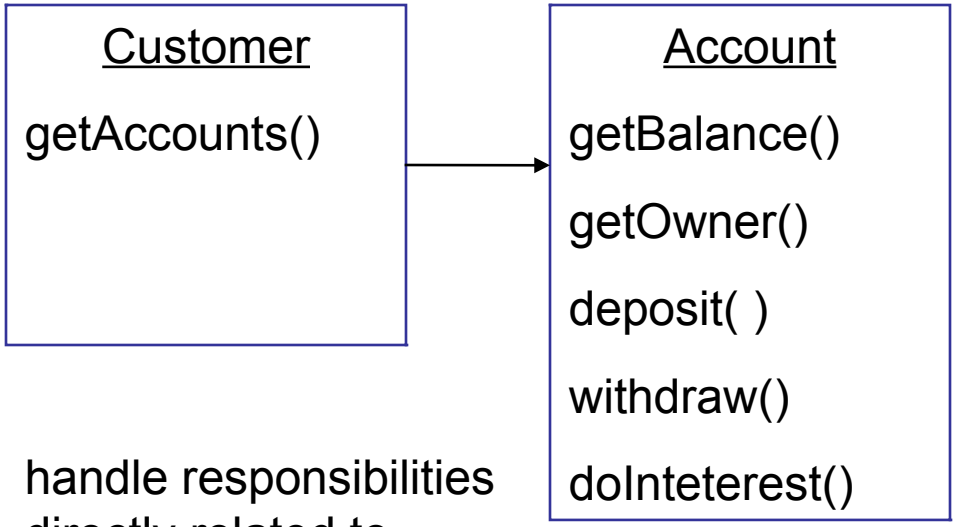


This has **high coupling** (bad)

- HARD TO MAINTAIN (MODIFY)
- HARD TO TEST

Cohesion

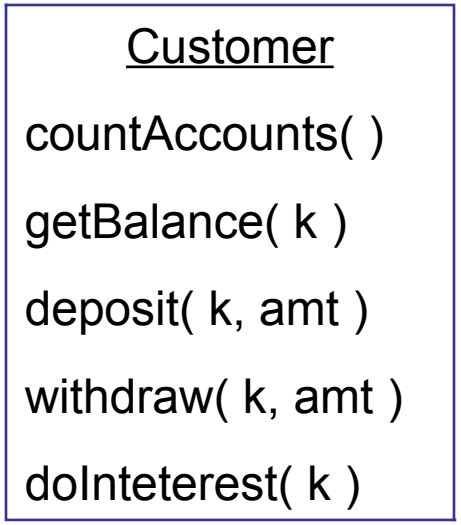
Good Design



handle responsibilities directly related to customer.

handle responsibilities directly related to bank accounts.

Bad Design



handle responsibilities directly related to bank accounts.

Consistency

Convert String to a Number:

- d = Double.parseDouble(string);
- k = Integer.parseInt string);
- l = Long.parseLong (string);

Convert Number to a String:

- s = Double.toString(d);
- s = Long.toString(l);
- s = Integer.toString(k);

Consistent Interface

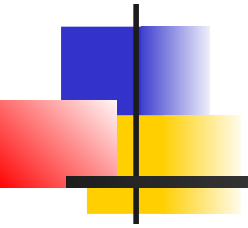
Convenience

```
s = Long.toString( n );
```

is the same as:

```
s = ( new Long( n ) ).toString();
```

which one is more *convenient?*



Modeling

3 Levels of Modeling

- Domain Model - model of the problem in terminology of the domain of application. Uses stakeholders' terms.

  Product, Product Catalog, Store, Sale

- Design Model - software design.

  Assign responsibilities

- Implementation Model - package, class, and behavior model for how the software will be implemented.

  Sale has a List<Product>, addToCart(...),

  removeFromCart(...)

4 Views of a Model

Different views to help understand the problem

1. Class Model - model of structure
2. State Model - the states of components and how they transition to other states
3. Behavioral Model - how objects react to different external triggers
4. Collaboration Model - how objects interact

Which UML diagram(s) are useful for these views?

1. Class diagram
2. State (Machine) diagram
3. Sequence and Activity diagrams
4. Communications Diagram