CS 535 Object-Oriented Programming & Design
Fall Semester, 2013
Doc 17 - Hinges
Nov 26 2013

Hinges

Tuesday, November 26, 13

http://en.wikipedia.org/wiki/File:Hamburgerpaumelle.JPG

Bank Account and Withdrawal

Type of changes

- New types of customers
- Change fee structure
- Change when to apply fee

```
BankAccount>>withdrawalNormal: aCurrency
    | newBalance |
    newBalance := balance - aCurrency.
    newBalance isNegative ifTrue: [
        balance := balance - 5.0 asCurrency.
        etc.


BankAccount>>withdrawalPreferred: aCurrency
    | newBalance |
    newBalance := balance - aCurrency.
    newBalance < -1000 asCurrency ifTrue: [
        balance := balance - 3.0 asCurrency.
        etc.
```

Adding New Types of Customers

Requires

New method in BankAccount

Callers need to be changed to call new method

```
BankAccount>>withdrawal: aCurrency
    | newBalance |
    newBalance := balance - aCurrency.
    balanceLimit := self isNormal
        ifTrue: [0 asCurrency]
        ifFalse: [-1000.0 asCurrency].
    overDraftFee := self isNormal
        ifTrue: [0 asCurrency]
        ifFalse: [-5.0 asCurrency].

newBalance < balanceLimit ifTrue: [
    balance := balance - overDraftFee.
    etc.
```

New Customer types, Fee & Limit Changes

Require

Editing the method

Other classes still call same method

Using instance variables for balanceLimit & overDraftFee

```
BankAccount>>withdrawal: aCurrency
    | newBalance |
    newBalance := balance - aCurrency.

    newBalance < balanceLimit ifTrue: [
        balance := balance - overDraftFee.
        etc.
```

New Customer types, Fee & Limit Changes

Just Data

Types & amounts could be read from file/database

Possible to

Create new customer types

Change fees

Change limits

without changing your code!

http://en.wikipedia.org/wiki/File:Hamburgerpaumelle.JPG

Second Example

```
SomeClass>>someMethod
    blah
    transaction = 'Withdrawal' ifTrue:[ account withdrawal: amount].
    blah.
    transaction = 'Deposit' ifTrue:[ account deposit: amount].
    etc
```

```
SameClassOrDifferentClass>>someOtherMethod
    blah
    transaction = 'Withdrawal' ifTrue:[ amount := data at: 3].
    blah.
    transaction = 'Deposit' ifTrue:[ amount := data at: 4].
    etc
```

Adding new Transactions

Require

Find all methods using transactions

Modifying each method

Hinge - Use Objects & Polymorphism

```
SomeClassOrDifferentClass>>someOtherMethod
    blah
    transaction = 'Withdrawal' ifTrue:[ amount := data at: 3].
    blah.
    transaction = 'Deposit' ifTrue:[ amount := data at: 4].
    etc
```

```
SomeClassOrDifferentClass>>someOtherMethod
    amount := transactionObject amount
```

Some Software hinges

Information Hiding
Encapsulation
Little pieces
Separation of Concerns
Abstractions
Once and only once
Polymorphism
Data files/databases
Design Patterns
Meta-data
Meta-Programming

Information Hiding & Encapsulation

Example 1 - Methods/functions

Moving an operation in to separate method/function

Isolates changes of operation to function

Example 2 - Currency

Some Currency classes have errors - need change

Little pieces

Smaller pieces if done well isolates more operations

Separation of Concerns

Like little pieces divides code to parts that can be changed independently

```
BankAccount>>fromTransactions: aStringOrFilename

	| stringTransactions |
	stringTransactions := aStringOrFilename asFilename contentsOfEntireFile.
	self addTransactions: stringTransactions asTransactions.
```

```
String>>asTransactions

	^self lines collect: [:each | BankTransaction from: each]
```

```
String>>lines

	^self tokensBasedOn: Character cr.
```

```
BankTransaction class>>from: aString

	| tokens tokensCleaned type |
	tokens := aString tokensBasedOn: Character tab.
	tokensCleaned := tokens collect: [:each | each trimSeparators].
	type := (tokensCleaned at: 3) asLowercase.
	self subclasses
		do: [:each | each type = type ifTrue: [^each components: tokensCleaned]]
```

```
BankTransaction class>>components: anArray

	^super new setComponents: anArray
```

```
BankTransaction >>setComponents: anArray

        self subclassResponsibility
```

BankTransaction Subclasses

Deposit class>>type

```
^'deposit'
```

NewAccount class>>type

```
^'newaccount'
```

Withdrawal class>>type

```
^'withdrawal'
```

How to Add New Transaction Types

Create subclass of BankTransaction

Subclass implements:

- Class method 'type'
- Instance method 'setComponents:'

BankAccount does not change

Code Reading transaction files does not change

Hinges and Design Decisions

Previous slides hard codes file structure across multiple classes/methods

- Carriage return
- tab

Solution is brittle regard to file format

Having an I/O related class would be better hinge

Polymorphism

```
BankAccount>>deposit: aTransaction
    blah

    aTransaction type = 'check' ifTrue: [blah ].
    aTransaction type = 'cash' ifTrue: [blah].
```

Adding new types of transaction requires changing this method

Polymorphism

```
BankAccount>>deposit: aTransaction

    newBalance := balance + aTransaction balanceAmount.
    newAvailableBalance := availableBalance +
        aTransaction availableBalanceAmount.
```

Adding new types of transaction does not requires changing this method

Unless new trasaction requires more complex operation

More Complex Polymorphism

```
BankAccount>>process: aTransaction

	aTransaction evaluteOn: self.
```

```
Deposit>>evaluteOn: aBankAccount

	aBankAccount deposit: self
```

```
Withdrawal>>evaluteOn: aBankAccount

	aBankAccount withdrawal: self
```

```
Cancel>>evaluteOn: aBankAccount

	originalTransaction := aBankAccount transactionAt: idToCancel.
	originalTransaction cancel.
	aBankAccount withdrawal: self
```

Business Rules

Some businesses frequently change rules/deals

- Buy two X and get third X for 1/2 price
- 20 cent coffee day

Don't have time to rewrite code

Need to move business logic into data