CSC407: Software Architecture
Fall 2006
Refactoring

Greg Wilson
BA 4234
gvwilson@cs.utoronto.ca

Announcements

- Seneca FSOSS
  - http://cs.senecac.on.ca/fsoss/2006/index.html
  - Oct 26-27, 2006, York University
- CUTC
  - http://www.cutc.ca/
  - Jan 11-13, 2007, in Toronto
- CUSEC
  - http://www.cusec.net/
  - Jan 18-20, 2007, in Montreal

Making Code Better

- The first lectures discussed principles that good object-oriented code should obey
- But what if the code you have doesn't?
- *Refactoring* is the process of reorganizing code to improve structure, or make room for future extension
  - Without changing functionality

Refactoring and Patterns

- Design patterns describe what code looks like when it's standing still
- Refactoring patterns describe how code changes over time
- Usually (but not often) you *refactor to patterns*
  - I.e., move from poor or inappropriate organization to one or more recognized patterns

Blackjack

- A card game involving one or more players and a dealer
  - Start with two cards, and can ask for more
  - Players go first; dealer goes last
    - Dealer must take a card when under 17
    - Must "stand" at 17 or over
  - Objective is to get as close to 21 as possible, *without* going over

A Blackjack Hand

```
public class BlackjackHand {
  private ArrayList<Card> cards = new Arraylist<Card>();
  public void addCard(Card card) {
    cards.add(card);
  }
  public List<Card> getCards() {
    return cards;
  }
  public boolean isBusted() {
    return getCount() > 21;
  }
  public int getResult(BlackjackDealer dealer) {
    // figure out if this hand beats the dealer
  }
  public int getCount() {
    // return the value of this hand
  }
  public String toString() { … }
}
```

Want to be able to re-use this in other card games

Why do we depend on the dealer to figure out if we won?

Cards

```
public class Card {
  public enum Suit { CLUBS, DIAMONDS, HEARTS, SPADES }
  public Suit suit;
  private int value;
  private BlackjackHand heldBy;
  public Card(Suit suit, int value) { … }
  public void setHand(BlackjackHand hand) {
    heldBy = hand;
  }
  public BlackjackHand getHand() {
    return heldBy;
  }
  public int getLowValue() {
    // value or 1 for ace, 10 for face cards
  }
  public int getHighValue() {
    // value or 11 for ace, 10 for face cards
  }
}
```

Won't be able to use this in a poker game

This is very specific to blackjack.

The Dealer

```
public class BlackjackDealer {
  private BlackjackHand hand = new BlackjackHand();
  public Card getUpCard() {
    return hand.getCards().get(1);
  }
  public BlackjackHand getHand() {
    return hand;
  }
  public boolean mustHit() {
    return hand.getCount() < 17;
  }
}
```

OK so far, but…

Players

```
public class ComputerBlackjackPlayer {
  private BlackjackHand hand = new BlackjackHand();
  public BlackjackHand getHand() { return hand; }
  public boolean wantCard(BlackjackDealer dealer) {
    // artificial intelligence goes here…
  }
}

public class HumanBlackjackPlayer {
  private BlackjackHand hand = new BlackjackHand();
  public BlackjackHand getHand() { return hand; }
  public boolean wantCard() {
    // user interface goes here…
  }
}
```

These two classes have nothing in common with each other, or with the dealer shown on the previous slide!

9

The Game

```
public class Blackjack {

  public static void main(String[] args) {
    Blackjack bj = new Blackjack();
    bj.playGame();
  }

  public void playGame() {
    ArrayList<Card> deck = new ArrayList<Card>(52);
    for (Card.Suit suit : Card.Suit.values()) {
      for (int value=1; value<=13; ++value) {
        Card card = new Card(suit, value);
        deck.add(card);
      }
    }

⇓
```

This block is not specific to blackjack: we should make it reusable

10

…The Game

```
  Collections.shuffle(deck);
  ComputerBlackjackPlayer computer = new ComputerBlackjackPlayer();
  HumanBlackjackPlayer human = new HumanBlackjackPlayer();
  BlackjackDealer dealer = new BlackjackDealer();
  for (int c=0; c<2; c++) {
    Card card = deck.remove(0);
    BlackjackHand computerHand = computer.getHand();
    computerHand.addCard(card);
    card.setHand(computerHand);
    card = deck.remove(0);
    BlackjackHand humanHand = human.getHand();
    humanHand.addCard(card);
    card.setHand(humanHand);
    card = deck.remove(0);
    BlackjackHand dealerHand = dealer.getHand();
    dealerHand.addCard(card);
    card.setHand(dealerHand);
  }
⇓
```

Deals two cards to each player—surely this can be simpler

11

…The Game

```
  System.out.println("Dealer has: " + dealer.getUpCard());
  boolean computerBusted = false;
  while (computer.wantCard(dealer)) {
    Card card = deck.remove(0);
    BlackjackHand computerHand = computer.getHand();
    computerHand.addCard(card);
    card.setHand(computerHand);
    if (computerHand.isBusted()) {
      computerBusted = true;
      break;
    }
  }
  if (computerBusted) {
    System.out.println("*** Computer busted! ***");
  }
  else {
    System.out.println("*** Computer stands ***");
  }
⇓
```

Scattered println statements as a user interface? Bleah…

And haven't we seen the lines that give players cards before?

12

…The Game

```
boolean humanBusted = false;
System.out.println("Your hand: " + human.getHand());
while (human.wantCard()) {
  …basically the same code as above…
}

boolean dealerBusted = false;
…basically the same code as above…
```

⇓

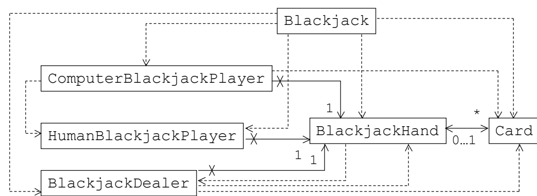
The three players play in almost the same way; the only things that differ are what they're called, and how they decide whether or not to take another card

…The Game

```
System.out.println("*** Results ***");
int compResult = computer.getHand().getResult(dealer);
if (compResult > 0) {
  System.out.println("Computer wins");
} else if (compResult < 0) {
  System.out.println("Computer loses");
} else {
  System.out.println("Computer pushes");
}
int humanResult = human.getHand().getResult(dealer);
if (humanResult > 0) {
  System.out.println("You win!");
} else if (humanResult < 0) {
  System.out.println("You lose :-(");
} else {
  System.out.println("You push");
}
```

More repetition, more embedded printing… (Oh, and "push" is blackjackese for "tie")

Just How Bad Is This?



Eww…

Problems

- Every `Card` knows about the `BlackjackHand` that's holding it
  – Which makes the `Card` class hard to re-use in a poker program
- The `Blackjack` class runs everything
  – Creating a deck of cards and shuffling it ought to be reusable

…Problems

- Parts of the `BlackjackHand` class could be used for other kinds of "hands"
- There's a *lot* of duplicated code
- `BlackjackHand` depends on `BlackjackDealer`
  - Because one of `BlackjackHand`'s methods takes a `BlackjackDealer` as a parameter

17

…Problems

- `BlackjackDealer` has one big monolithic `playGame` method
  - Quick, tell me how you're going to test it
- The UI consists of scattered print statements
  - How are we going to build a GUI for this, or a web interface?

18

Step 1: Deblackjackification

```
public class Card {
  public enum Suit { CLUBS, DIAMONDS, HEARTS, SPADES }
  public Suit suit;
  private int value;
  private BlackjackHand heldBy;
  public Card(Suit suit, int value) { … }
  public void setHand(BlackjackHand hand) {
    heldBy = hand;
  }
  public BlackjackHand getHand() {
    return heldBy;
  }
  public int getLowValue() {
    // value or 1 for ace, 10 for face cards
  }
  public int getHighValue() {
    // value or 11 for ace, 10 for face cards
  }
}
```

References to BlackjackHand

Other games don't give cards these values (or distinguish low and high values)

19

A Non-Solution

- Option 1: create an abstract Hand base class, and derive BlackjackHand from that
- But that still leaves this fragile code:

```
computerHand.addCard(card);
card.setHand(computerHand);
```

- Do cards really need to know what hands their in?

20

Our First Refactoring Pattern

- *Change Bidirectional Association to Unidirectional Association*
  - BDAs aren't intrinsically bad
  - But they *are* intrinsically harder to manage
- Remove `getHand` and `setHand` from `Card`, remove the `setHand` calls in `Blackjack`
  - …nothing breaks!

21

Moving Methods

- `getLowValue` and `getHighValue` are next
  - Can't delete them, since `getCount` needs the information they provide
- Options:
  - Move into an existing subclass
  - Create a subclass and move the methods down

22

Spotting Movable Methods

- If a method uses another object more than it uses itself, it's a candidate for moving

```
public class X {
  public int meth(Y y) {
    int y1 = y.one();
    int y2 = y.two();
    int y3 = y.three();
    return y1 + y2 + y3;
  }
}
```

```
public class Y {
  public int meth() {
    return one() + two() + three();
  }
}
```

23

What About Subclassing?

- The other option is to the logic into a new subclass
  - Have `Card.getValue` return the "raw" value
  - Have `BlackjackCard` wrap `getValue` in two new methods `getLowValue` and `getHighValue`
  - Then change references from `Card` to `BlackjackCard`
- Votes?

24

Open-Closed Principle

```
public Deck() {
  cards = new ArrayList<Card>(52);
  for (Card.Suit suit : Card.Suit.values()) {
    for (int value=1; value<=13; ++value) {
      Card card = new Card(suit, value);
      cards.add(card);
    }
  }
}
```

```
public Deck() { cards = createCards(); }
protected List<Card> createCards() {
  List<Card> cards = new ArrayList<Card>(52);
  for (Card.Suit suit : Card.Suit.values) {
    for (int value=1; value<=13; ++value) {
      Card card = createCard(suit, value);
      cards.add(card);
    }
  }
  return cards;
}
protected Card createCard(Card.Suit s, int v) {
    return new Card(s, v);
}
```


Why Is This Better?

- Most of the code looks the same
- But:
  - It'll be easier to create decks of 48 cards (for pinochle)
  - There's an obvious place to create a `BlackjackCard` instead of a basic `Card`
- Question: what about the 52 and 13?

Open For Extension

- Let's make sure that the blackjack program creates instances of `BlackjackCard`

```
public class BlackjackDeck extends Deck {
  protected Card createCard(Card.Suit s, int v) {
    return new BlackjackCard(s, v);
  }
  public BlackjackCard dealCard() {
    return (BlackjackCard)super.dealCard();
  }
}

public class Deck {
  public Card dealCard() {
    return cards.get(nextCard++);
  }
}
```

No need to override the createCard method

Look up the meaning of "covariant return types" before next class

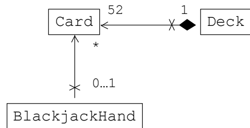
Impact

```
public void playGame() {
  ArrayList<Card> deck = new ArrayList<Card>(52);
  for (Card.Suit suit : Card.Suit.values()) {
    for (int value=1; value<=13; ++value) {
      Card card = new Card(suit, value);
      deck.add(card);
    }
  }
  Collections.shuffle(deck);
  …create players…
  for (int c=0; c<2; c++) {
    Card card = deck.remove(0);
    BlackjackHand computerHand = computer.getHand();
    computerHand.addCard(card);
    card = deck.remove(0);
```

```
BlackjackDeck deck =
  new BlackjackDeck();
deck.shuffle();
```

```
BlackjackCard card =
  deck.dealCard();
```

To Put It Another Way…



Our Next Target

```
public class BlackjackHand {

  private ArrayList<Cards> cards = new ArrayList<Cards>;
  public void addCard(Card card) { cards.add(card); }
  public List<Card> getCards() { return cards; }

  public boolean isBusted() { return getCount() > 21; }
  public int getResult(BlackjackDealer dealer) {
    // compute winner
  }
  public int getCount() {
    // figure out what the hand's worth
  }

  public String toString() { … }
}
```

Reusable

Specific to blackjack

Partially reusable?

Yet More Abstraction

```
public abstract class Hand {
  private List<Card> cards = new ArrayList<Card>();

  public void addCard(Card card) { cards.add(card); }

  public List<Card> getCards() {
    return Collections.unmodifiableList(cards);
  }

  public toString() { return cards.toString(); }

  public void clear() { cards.clear(); }
}
```

Look up what this does for next week

We can probably improve this

Since we're here…

…Yet More Abstraction

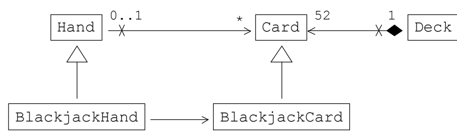
```
public class BlackjackHand extends Hand {
  public boolean isBusted() {
    return getCount() > 21;
  }
  public int getResult(BlackjackDealer dealer) {
    // compute hand winner
  }
  public int getCount() {
    // figure out what hand's worth
  }
  public String toString() {
    return super.toString() + ": " + getCount();
  }
  public Card getUpCard() {
    return getCards().get(1);
  }
}
```

No longer explicitly managing cards

The improved version

Since we're here…

The Extended Design



33

Half the Changes

- We've now fixed the problems with cards, decks, and hands
- Still have to fix problems with players
  - Human, computer, and dealer overlap in many ways
- http://www.refactoring.com/ is required reading for this course

34

Questions?

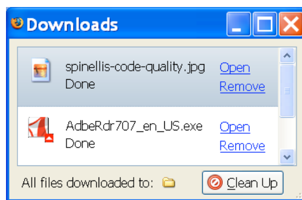


Fig. 1: Dialog (front view)

Fig. 2: Dialog (side view)

35