Introduction to Object Oriented Programming 2E Timothy A. Budd

Chapter 11

Replacement and Refinement

Replacement and Refinement

What happens when child classes have methods with the same names as parent?

There are two general models describing this situation:

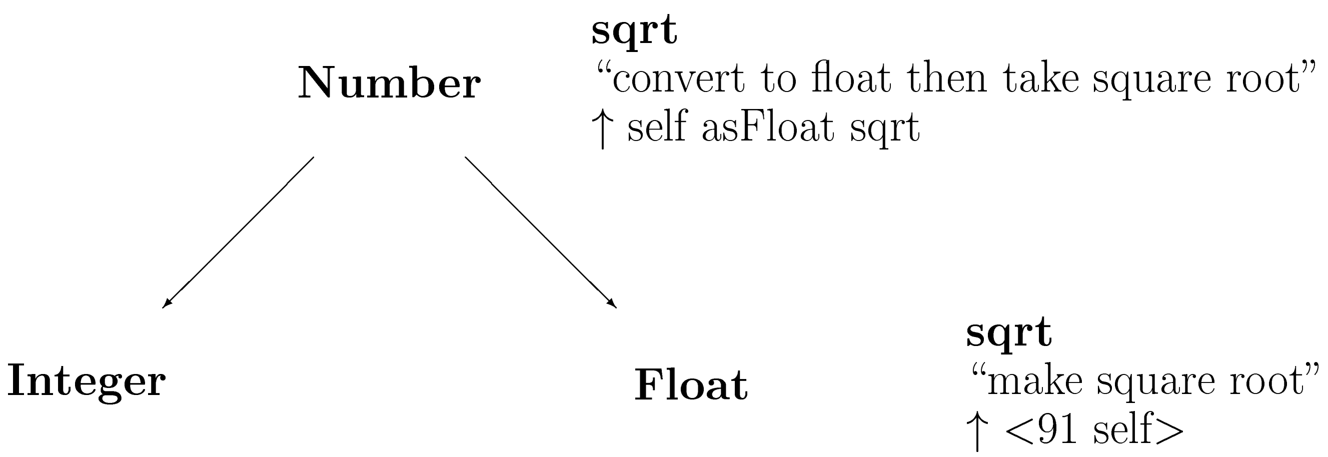
Replacement: "American" school of OOP – child method replaces parent method

Refinement: "Scandinavian" school of OOP – behavior of parent and child classes are merged to form new behavior

If replacement used, then some mechanism usually provided for refinement

If refinement default, then replacement not possible

Replacement Example – sqrt in Smalltalk



Class **Float** replaces the method for square root.

If not replaced (for example, for integers) then the method in class **Number** is used.

Preserving is-a

Replacement makes preserving principle of substitutability difficult

- No guarantee that child class will do anything similar to the parent
- Other methods that depend upon behavior from parent class may fail
- Great havoc can ensue if programmer is not careful

Documenting Replacement or Refinement

- Some languages require parent class to indicate replacement is permitted (e.g., virtual in C++)
- Some languages require child class to indicate replacement (e.g., override in Object Pascal)
- Some languages do not automatically document replacement or refinement (e.g., Smalltalk, Java)

Replacement in C++ – Issues

Replacement in C++ is complicated

- If method declared virtual, then polymorphic
  - binding based on dynamic class
- If method not declared **virtual**, then method not polymorphic
  - binding based on static class
- Constructors use refinement; all other methods use replacement

Replacement in C++ – Class Descriptions

```
class CardPile {
public:
    ...
    virtual int canTake(Card *);
    ...
};

class SuitPile : public CardPile {
public:
    ...
    virtual int canTake(Card *);
    ...
};
```


Replacement in C++ – Methods

```
int CardPile::canTake(Card * aCard)
{   // just say no
    return 0;
}

int SuitPile::canTake(Card * aCard)
{
    if (top == nilLink) {   // empty - can take ace
        if (aCard->rank() == 1) return 1;
        return 0;
        }
    if ((top->card())->suit() != aCard->suit())
        return 0;
    if (((top->card())->rank() + 1) == aCard->rank())
        return 1;
    return 0;
}
```


Replacement in Java

- Similar to replacement in C++, but no virtual keyword
- Can even replace data fields, although these are static and not dynamic
- Keyword final on method prohibits replacement
- Keyword final on class prohibits subclassing altogether

Replacement in Objective-C and Smalltalk

- Occurs whenever method in child class has same name as in parent

- Square root method in Smalltalk presented earlier

Refinement

- Method in parent and method in child merged to form new behavior
- Parent class method *always* executed
- Users guaranteed at least behavior of parent
- Performed automatically in some languages (Simula, Beta)
- Performed by directly invoking parent method from within child method in other languages

Refinement in Simula and Beta

- Execute code from parent first – when INNER statement encountered, execute child code
- Parent classes *wrap* around child – almost reverse of American school languages
- Guarantees functionality provided by parent classes

Refinement in C++

```
// place a single card into a card pile
void CardPile::addCard(Card * aCard)
{
    if (aCard != nilcard) {
        aCard->setLink(top);
        top = aCard;
        top->moveTo(x, y);
        }
}

// place a single card into a table pile
void TablePile::addCard(Card * aCard)
{   int tx, ty;

    // pile is empty, just place it normally
    if (top == nilcard)
        CardPile::addCard(aCard);  // note explicit call
    else {    // display it below other cards
        tx = top->locx();
```

```
        ty = top->locy();
        // figure out where to place card
        if (top->faceUp() && top->next() != nilcard &&
            (top->next())->faceUp())
                ; // do nothing
        else
            ty += 30;    // move down a bit
        CardPile::addCard(aCard);  // note explicit call
        top->moveTo(tx, ty);
        }
}
```

Constructors Always Do Refinement

```
class TablePile : public CardPile {
pubic:
    TablePile(int, int, int);
    ...
private:
    int column;
};

TablePile::TablePile(int c, int x, int y) : CardPile(x, y)
{
    column = c;
}
```


Refinement in Java

```
class CardPile
{   CardPile (int xl, int yl) { x = xl; y = yl; cardList = new LinkedList(); }

    public void addCard (Card aCard) { cardList.add(aCard); }
   ...
}

class DiscardPile extends CardPile
{   DiscardPile (int x, int y) { super (x, y); }  // parent's constructor

    public void addCard (Card aCard)
    {   if (! aCard.faceUp())
            aCard.flip();
        super.addCard(aCard);  // parent's addCard method
    }
   ...
}
```