Inheritance and Delegation

Delegation of behaviour

- Behaviour can be spread across multiple classes.
- Classes can delegate (part of) their behaviour to other classes
  - To inheritance parent
    - with a method call to self
  - To inheritance child
    - with a method call to self
  - To unrelated class
    - with a message call to a delegate object

Up calls

- (Note on directions: Going from specialized classes to general classes is said to be going “up” the inheritance hierarchy, regardless of how the class diagrams are drawn.)
- Consider two related classes:

```
class A extends C { …
    public void method() {
        W0
        X
        Y0 } … }
class B extends C { …
    public void method() {
        W1
        X
        Y1 } … }
```

Up calls (cont)

- The common code X can be factored into the parent

```
class C { …
    protected void helperMethod() {
        X } … }
class A extends C { …
    public void method() {
        W0
        helperMethod() ;
        Y0 } … }
class B extends C { …
    public void method() {
        W1
        helperMethod() ;
        Y1 } … }
```

Up Calls (cont.)

- In some cases the factored code may be the parent's implementation of the same method.

```
class C { …
    public void method() {
        X } … }
class A extends C { …
    public void method() {
        W0
        super.method() ;
        Y0 } … }
class B extends C { …
    public void method() {
        W1
        super.method() ;
        Y1 } … }
```

Up calls to constructors

- For constructors, Java (and C++) have special syntax for calling parent constructors.
- Java

```
class Parent { …
    protected Parent( params ) {
        initialization of parent's state } … }
class Child { …
    public Child( params ) {
        super( args ) ; // explicit call of parent's constructor
        additional initialization of child's state } … }
```

Down Calls (template method)

- Suppose that the following pattern occurs in child classes

```
class Child extends Parent { …
    public void someMethod() {
        local variable declarations
        common code 0
        code special to this class
        common code 1 } … }
```

  - We could create two helper methods in the parent
  - But if the two common parts share local variables, this is very awkward and requires child to declare and route the variables.

Down calls (Template Calls)

- The Template pattern provides a solution

```
class Parent { …
    public void someMethod() {
        local variable declarations
        common code 0
        hookMethod() ;
        common code 1 }
    protected void hookMethod() {
        default implementation or } … }
class Child extends Parent { …
    protected void hookMethod() {
        code special to this class } … }
```

  - Of course the hookMethod and hence Parent could be abstract.

Example javax.swing Painting

- In the library

```
class JComponent extends java.awt.Container { …
    public void paint(Graphics g) {
        paintComponent( g )  ;
        for each child, c, call c.paint(g) }
    protected void  paintComponent( Graphics g ) {
        /* do nothing */  } … }
```

- In client code

```
class SomeComponent extends JComponent { …
    protected void paintComponent( Graphics g ) {
        code special to this class } … }
```

Delegation

- Delegating to parent (up call) is convenient but.
  - What if common code appears in unrelated classes?
  - In this case we can delegate the common work to a helper class.

Example: The Observer Pattern

- Library class (common code)

```
public class ObserverHelper {
    private List<Observer> listOfObservers
                                    = new ArrayList< Observer>() ;
    public void notify() {
        for(Observer observer : listOfObservers) {
            observer.update() ; } }
    public void addObserver( Observer observer ) {
        listOfObservers.add( observer ) ; }
    public void removeObserver( Observer observer ) {
        listOfObservers.remove( observer ) ; } }
```

Example (cont)

- The client code

```
class ConcreteSubject extends Something {
    private ObserverHelper helper = new ObserverHelper() ;
        public void addObserver( Observer obs ) {
        helper.addObserver( obs ) ; }
    public void removeObserver( Observer obs ) {
        helper.addObserver( obs ) ; }
    private void notify() { helper.notify() ; }
    … }
```

  - Compare this to the GoF version of the Observer, which uses inheritance from an AbstractSubject rather than delegation.

Varying the helper class: Strategy pattern

"You can change your friends, but not your relations"

- While a class's parent class is fixed, the concrete class of a helper object can be determined an construction time or later.

```
class ConfigurableClass  {
    private HelperInterface helper;
    // Constructor
    public ConfigurableClass (HelperInterface helper) {
        … this.helper = helper ; …}
    public void changeHelper(HelperInterface helper ) {
        this.helper = helper ; }
    public void someMethod( ) {
        … helper.someHelperMethod() ; … } … }
```

Example: Layout managers in AWT

- In the AWT each Container object has a LayoutManager object.
  - LayoutManager is an interface

    void addLayoutComponent(String name, Component comp)

    void layoutContainer(Container parent)

    // ^^^ does the layout of the given container.

    void removeLayoutComponent(Component comp)

    Dimension minimumLayoutSize(Container parent)

    Dimension preferredLayoutSize(Container parent)
  - Containers use their layout object to determine where to place child Components.

Example: Layout managers in AWT

- Example layout managers in the AWT library
  - FlowLayout --- puts component in horizontal lines (like words in a paragraph).
  - GridLayout --- puts components in a grid
  - GridBagLayout --- a more flexible grid
  - BorderLayout --- puts one component in the middle and others around the edges.
- Mix and match
  - Library layout managers may be used with library or custom containers
  - Custom layout managers may be used with library or custom containers