Winter Semester 16/17

Software Engineering Design & Construction

Dr. Michael Eichberg
Fachgebiet Softwaretechnik
Technische Universität Darmstadt

A Critical View on Inheritance

A Critical View On Inheritance

Inheritance is *the main* built-in variability mechanism of OO languages.

Desired Properties

(Of Programming Languages)

- Built-in support for OCP
- Good Modularity
- Support for structural variations
- Variations can be represented in type declarations

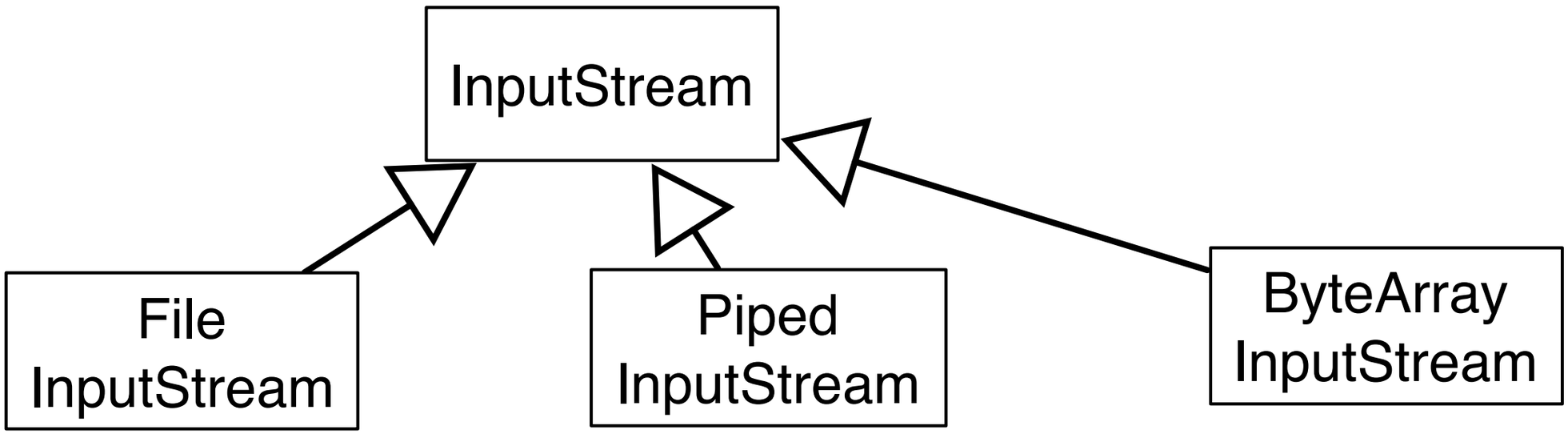
Variation of selection functionality of table widgets.

Desired Properties By Example

```
class TableBase extends Widget {
  TableModel model;
  String getCellText(int row, int col){ return model.getCellText(row, col); }
  void paintCell(int r, int c){ getCellText(row, col) … }
}
abstract class TableSel extends TableBase {
  abstract boolean isSelected(int row, int col);
  void paintCell(int row, int col) { if (isSelected(row, col)) … }
}
class TableSingleCellSel extends TableSel {
  int currRow; int currCol;
  void selectCell(int r, int c){ currRow = r; currCol = c; }
  boolean isSelected(int r, int c){ return r == currRow && c == currCol; }
}
class TableSingleRowSel extends TableSel {
  int currRow;
  void selectRow(int row) { currRow = row; }
  boolean isSelected(int r, int c) { return r == currRow; }
}
class TableRowRangeSel extends TableSel { … }
class TableCellRangeSel extends TableSel { … }
```

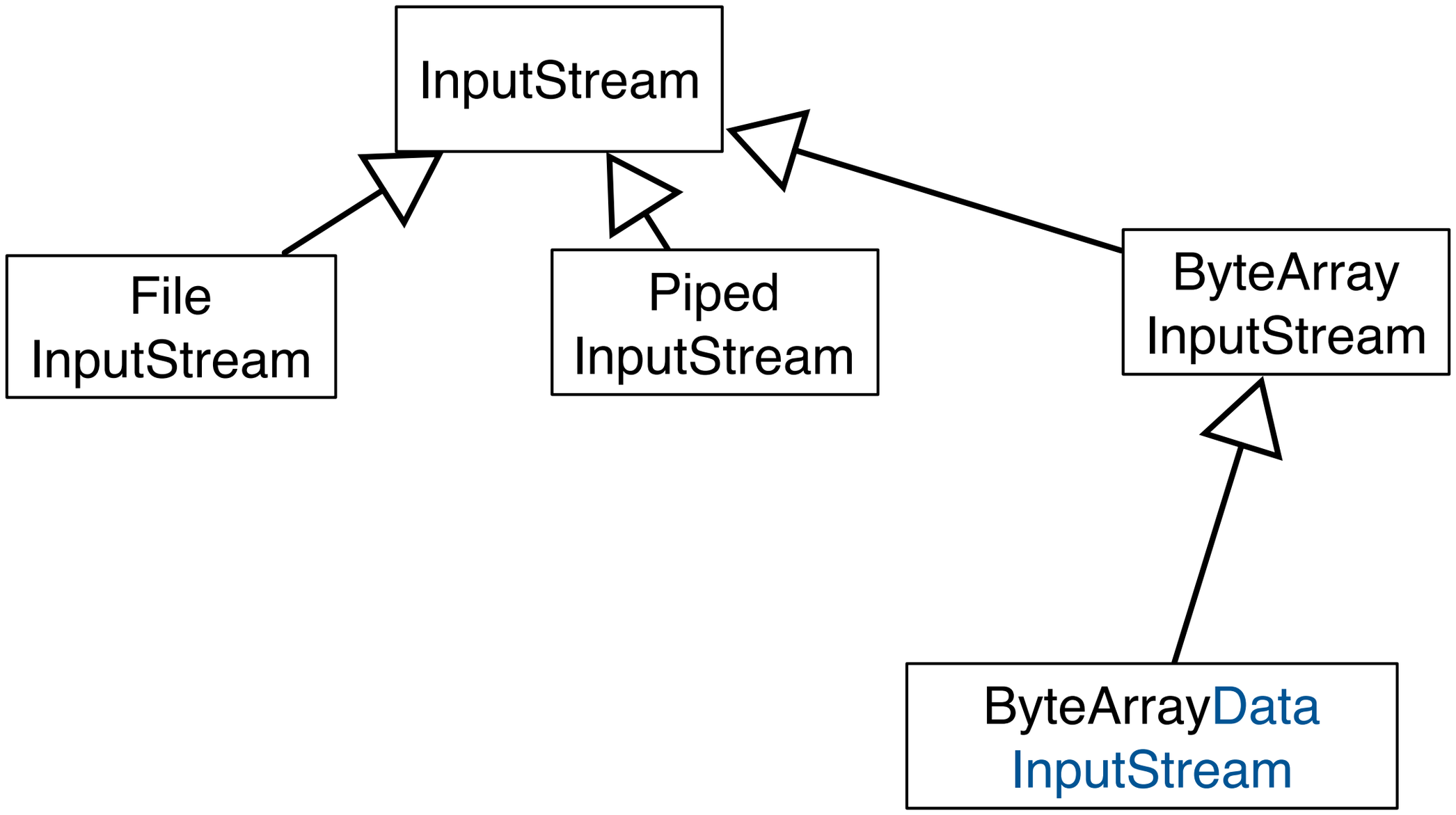
Non-Reusable, Hard-to-Compose Extensions

An Extract from Java's Stream Hierarchy



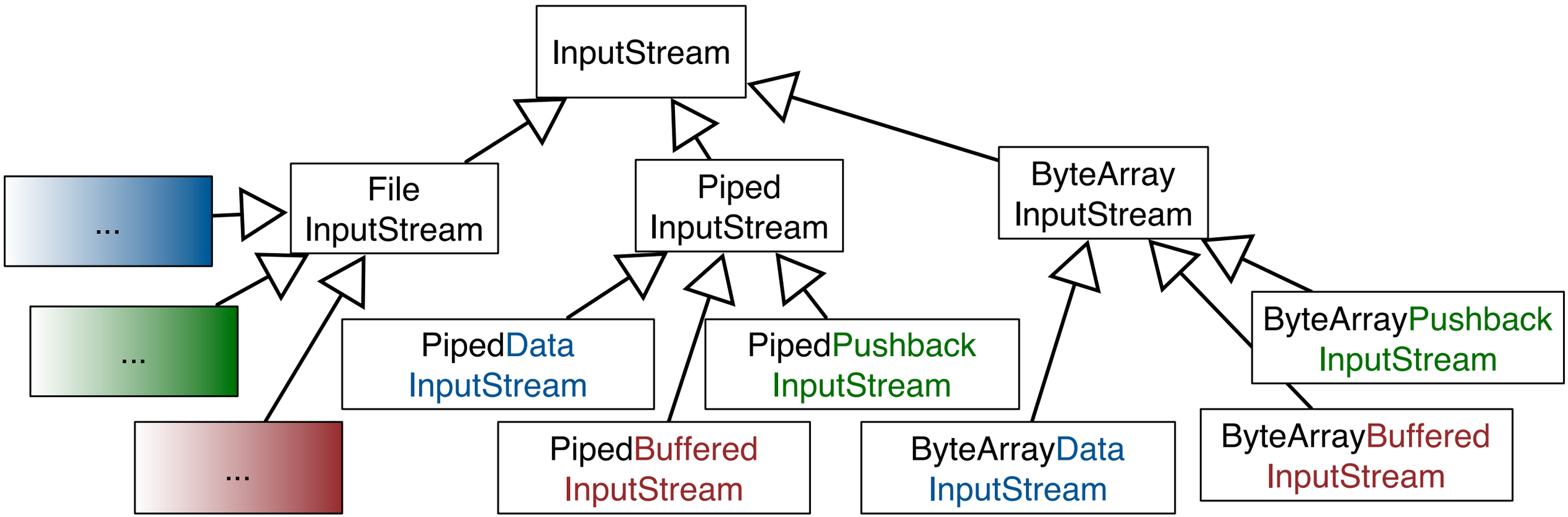
Non-Reusable, Hard-to-Compose Extensions

An Extract from Java's Stream Hierarchy - A Simple Variation



Non-Reusable, Hard-to-Compose Extensions

An Extract from Java's Stream Hierarchy - A Simple Variation



Each kind of variation would have to be re-implemented for all kinds of streams, for all meaningful combinations of variations

Non-Reusable, Hard-to-Compose Extensions

Extensions defined in subclasses of a base class cannot be reused with other base classes.

The Pushback related functionality in `FilePushbackInputStream` cannot be reused.

Weak Support for Dynamic Variability

Variations supported by an object are fixed at object creation time and cannot be (re-)configured dynamically.

A buffered stream is a buffered stream is a buffered stream… It is not easily possible to turn buffering on/off if it is implemented by means of subclassing.

Dynamic Variability Illustrated

The configuration of an object's implementation may depend on values from the runtime context.

- Potential Solution:
  Mapping from runtime values to classes to be instantiated can be implemented by conditional statements.
  …`if(x) new Y() else new Z()` …
- **Issue**:
  Such a mapping is error-prone and not extensible. When new variants of the class are introduced, the mapping from configuration variables to classes to instantiate must be changed.

Dynamic Variability Illustrated

The configuration of an object's implementation may depend on values from the runtime context.

- Potential Solution:
  Using dependency injection.

- Issue:
  Comprehensibility suffers:

  - Objects are (implicitly) created by the Framework
  - Dependencies are not directly visible/are rather implicit

Dynamic Variability Illustrated

The behavior of an object may vary depending on its state or context of use.

- Potential Solution:
  Mapping from runtime values to object behavior can be implemented by conditional statements in the implementation of object's methods.

- **Issue**:
  Such a mapping is error-prone and not extensible.
  When new variants of the behavior are introduced, the mapping from dynamic variables to implementations must be changed.

The Fragile Base Class Problem

Cf. Item 17 of Joshua Bloch's, **Effective Java**.

An Instrumented HashSet

The Fragile Base Class Problem Illustrated

```
import java.util.*;
public class InstrumentedHashSet<E> extends HashSet<E> {
  private int addCount = 0;
  public InstrumentedHashSet() {    }
  public InstrumentedHashSet(int initCap, float loadFactor) {
    super(initCap, loadFactor);
  }
  @Override public boolean add(E e) { addCount++; return super.add(e); }
  @Override public boolean addAll(Collection<? extends E> c) {
    addCount += c.size();
    return super.addAll(c);
  }
  public int getAddCount() { return addCount; }

  public static void main(String[] args) {
    InstrumentedHashSet<String> s = new InstrumentedHashSet<String>();
    s.addAll(Arrays.asList("aaa", "bbb", "ccc"));
    System.out.println(s.getAddCount());
  }
}
```

Output?

An Instrumented HashSet

The Fragile Base Class Problem Illustrated

```
import java.util.*;
public class InstrumentedHashSet<E> extends HashSet<E> {
  private int addCount = 0;
  public InstrumentedHashSet() {    }
  public InstrumentedHashSet(int initCap, float loadFactor) {
    super(initCap, loadFactor);
  }
  @Override public boolean add(E e) { addCount++; return super.add(e); }
  // @Override public boolean addAll(Collection<? extends E> c) {
  //  addCount += c.size();
  //  return super.addAll(c);
  // }
  public int getAddCount() { return addCount; }

  public static void main(String[] args) {
    InstrumentedHashSet<String> s = new InstrumentedHashSet<String>();
    s.addAll(Arrays.asList("aaa", "bbb", "ccc"));
    System.out.println(s.getAddCount());
  }
}
```

Does this really(!) solve the problem?

The Fragile Base Class Problem in a Nutshell

Changes in base classes may lead to unforeseen problems in subclasses.

Inheritance Breaks Encapsulation

Fragility by dependencies on the self-call structure

The Fragile Base Class Problem in a Nutshell

- The fragility considered so far is caused by dependencies on the self-call structure of the base class.

- Subclasses make assumptions about the calling relationship between `public` and `protected` methods of the base class.

- These assumptions are implicitly encoded in the overriding decisions of the subclass.

- If these assumptions are wrong or violated by future changes of the structure of superclass' self-calls, the subclass's behavior is broken.

I/II

Fragility by addition of new methods

The Fragile Base Class Problem in a Nutshell

- Fragility by **extending a base class with new methods** that were not there when the class was subclassed.

- Example
  - Consider a base collection class.
  - To ensure some (e.g., security) property, we want to enforce that all elements added to the collection satisfy a certain predicate.
  - We override every method that is relevant for ensuring the security property to consistently check the predicate.
  - Yet, the **security may be defeated unintentionally** if a new method is added to the base class which is relevant for the (e.g., security) property.

Fragility by addition of new methods

The Fragile Base Class Problem in a Nutshell

- Fragility by **extending a base class with a method** that was also added to a subclass. I.e., we accidentally capture a new method; the new release of the base class accidentally includes a method with the same name and parameter types.

- If the return types differ, your code will not compile anymore because of conflicting method signatures.

- If the signature are compatible, your methods may get involved in things you never thought about.

II/III

Fragility by addition of new methods

The Fragile Base Class Problem in a Nutshell

- Fragility by **extending a base class with a method** that was also added to a subclass. I.e., we accidentally capture a new method; the new release of the base class accidentally includes a method with the same name and parameter types.

- If the return types differ, your code will not compile anymore because of conflicting method signatures.

- If the signature are compatible, your methods may get involved in things you never thought about.

III/III

Fragility by addition of new methods

The Fragile Base Class Problem in a Nutshell

- Fragility by **extending a base class with an overloaded method**; the new release of the base class accidentally includes a method which make it impossible for the compiler to determine the call target of your call.

```
class X { void m(String){…} ; void m(Object o){…}/*added*/ }

<X>.m(null)  // the call target is not unique (anymore)
```

Taming Inheritance

Implementation inheritance (`extends`) is a powerful way to achieve code reuse.

But, if used inappropriately, it leads to fragile software.

Dos and Don'ts

Taming Inheritance

Closed Packages Assumption

- It is *always* safe to use inheritance within a package. The subclass and the superclass implementation are under the control of the same programmers.

- It is also OK to **extend classes specifically designed and documented for extension.**

- Avoid inheriting from concrete classes not designed and documented for inheritance across package boundaries.

Design and document for inheritance or else prohibit it.

-Joshua Bloch, Effective Java

Classes Must Document Self-Use

Taming Inheritance

- Each public/protected method/constructor must indicate self-use:
  - Which overridable methods it invokes.
  - In what sequence.
  - How the results of each invocation affect subsequent processing.
- A class must document any circumstances under which it might invoke an overridable method. (Invocations might come from background threads or static initializers.)

Common Conventions for Documenting Self-Use

Taming Inheritance

- The description of self-invocations to overridable methods is given at the end of a method's documentation comment.
- The description starts with "`This implementation …`". Indicates that the description tells something about the internal working of the method.

Example of Documentation On Self-Invocation

- Taken from: `java.util.AbstractCollection`

```
public boolean remove(Object o)

Removes a single instance of the specified element from this
collection.
…
This implementation removes the element from the collection
using the iterator's remove method.
Note that this implementation throws an
UnsupportedOperationException if the iterator returned by
this collection's iterator() method does not implement the
remove(…) method.
```

Documenting Self-Use In API Documentation

Do implementation details have a rightful place in a good API documentation?

Example of Documentation On Self-Invocation

- Taken from: `java.util.AbstractList`

```
protected void removeRange(int fromIndex, int toIndex)

Removes from a list …

This method is called by the clear operation on this list and
its sub lists. Overriding this method to take advantage of the
internals of the list implementation can substantially improve
the performance of the clear operation on this list and its sub
lists…

This implementation gets a list iterator positioned before
fromIndex and repeatedly calls ListIterator.next and
ListIterator.remove. Note: If ListIterator.remove requires
linear time, this implementation requires quadratic time.
```

Carefully Design and Test Hooks To Internals

- Provide as few protected methods and fields as possible.

- Each of them represents a commitment to an implementation detail.

- Designing a class for inheritance places limitations on the class.

- Do not provide too few hooks.

- A missing protected method can render a class practically unusable for inheritance.

Constructors Must Not Invoke Overridable Methods

Constructors Must Not Invoke Overridable Methods

```
class JavaSuper {
  public JavaSuper() { printState(); }

  public void printState() { System.out.println("no state"); }
}

class JavaSub extends JavaSuper {
    private int x = 42; // the result of a tough computation

    public void printState() { System.out.println("x = " + x); }
}

class JavaDemo {
    public static void main(String[] args) {
        JavaSuper s = new JavaSub();
        s.printState();
    }
}
```

Result:

x = 0

x = 42

Constructors Must Not Invoke Overridable Methods

```
class ScalaSuper {

    printState(); // executed at the end of the initialization

    def printState() : Unit = { println("no state”) }
}
```

```
class ScalaSub extends ScalaSuper {
    var y: Int = 42 // What was the question?
    override def printState() : Unit = { println("y = "+y) }
}

object ScalaDemo extends App {
    val s = new ScalaSub
    s.printState() // after initialization
}
```

Result:

y = 0

y = 42

Constructors Must Not Invoke Overridable Methods

```
class Super {

    // executed at the end of the initialization
    printState();

    def printState() : Unit = { println("no state") }
}

class Sub(var y: Int = 42) extends Super {
    override def printState() : Unit = { println("y = "+y) }
}

object Demo extends App {
    val s = new Sub
    s.printState() // after initialization
}
```

Result:

y = 42

y = 42

Initializers Must Not Invoke *Overridable Methods*

```
trait Super {
    val s: String
    def printState() : Unit = { println(s) }

    printState();
}
```

```
class Sub1 extends Super { val s: String = 110.toString }
class Sub2 extends { val s: String = 110.toString } with Super
```

```
new Sub1()
new Sub2()
```

Result:
null
110

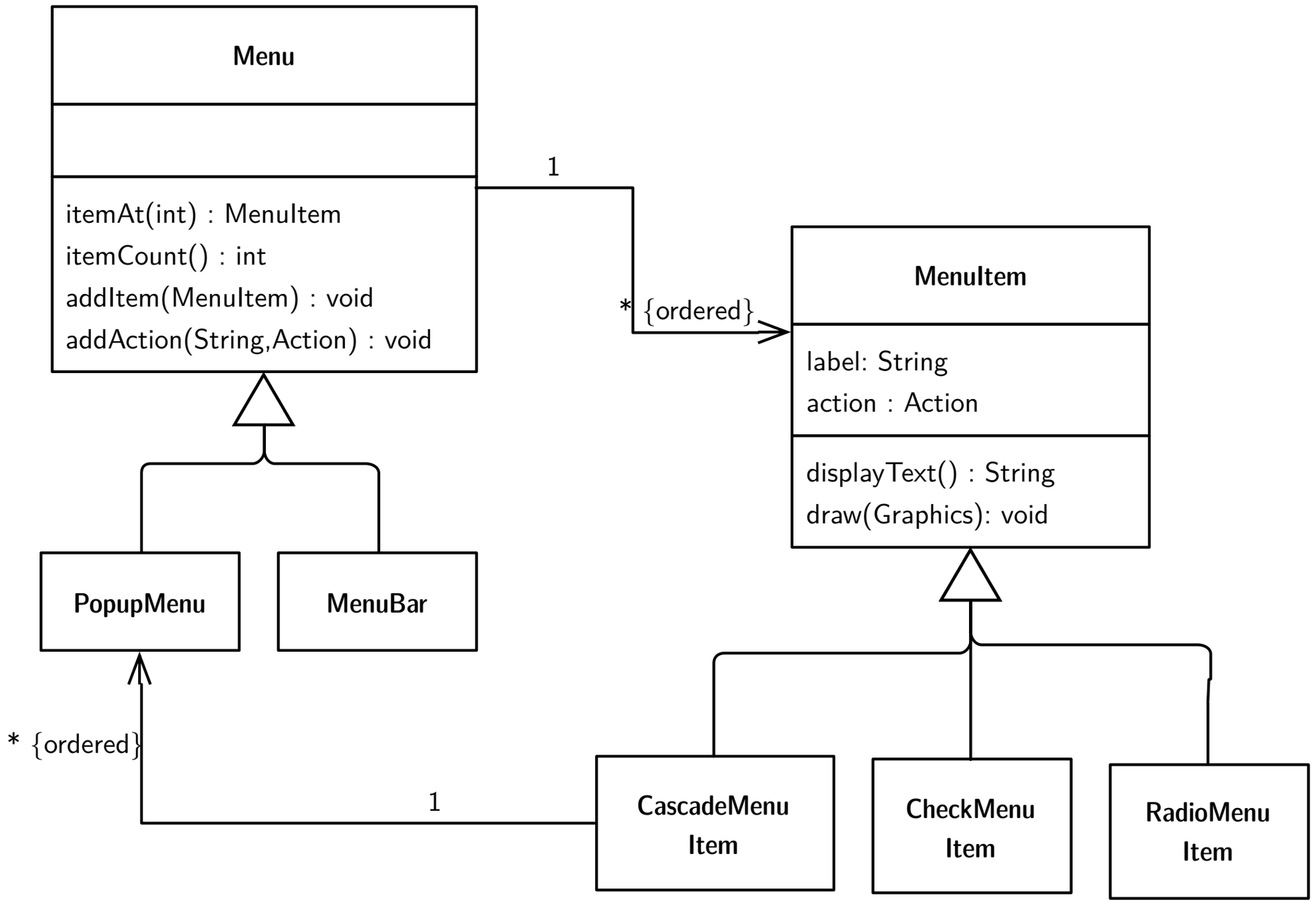
Variations at the Level of Multiple Objects

So far, we considered variations, whose scope are individual classes.

But, no class is an island!

Window Menus

Illustrative Example



Different Kinds of Menus

```
abstract class Menu {
  List<MenuItem> items;

  MenuItem itemAt(int i) {
    return items.get(i);
  }

  int itemCount() { return items.size(); }
  void addItem(MenuItem item) { items.add(item); }
  void addAction(String label, Action action) {
    items.add(new MenuItem(label, action));
  }
   ...
}

class PopupMenu extends Menu { ... }

class MenuBar extends Menu { ... }
```

Different Kinds of Menu Items

```
class MenuItem {
  String label;
  Action action;

  MenuItem(String label, Action action) {
    this.label = label;
    this.action = action;
  }

  String displayText() { return label; }

  void draw(Graphics g) { … displayText() … }
}

class CascadeMenuItem extends MenuItem {
  PopupMenu menu;

  void addItem(MenuItem item) { menu.addItem(item); }
  …
}

class CheckMenuItem extends MenuItem { … }

class RadioMenuItem extends MenuItem { … }
```

Inheritance for Optional Features of Menus

- Variations of menu functionality affect multiple objects constituting the menu structure.

- Since these objects are implemented by different classes, we need several new subclasses to express variations of menu functionality.

- **This technique has several problems**, which will be illustrated in the following by a particular example variation: Adding accelerator keys to menus.

Menu Items with Accelerator Keys

```
class MenuItemAccel extends MenuItem {
  KeyStroke accelKey;

  boolean processKey(KeyStroke ks) {
    if (accelKey != null && accelKey.equals(ks)) {
      performAction();
      return true;
    }
    return false;
  }

  void setAccelerator(KeyStroke ks) { accelKey = ks; }

  void draw(Graphics g) {
    super.draw(g);
    displayAccelKey();
  }
  …
}
```

Menus with Accelerator Keys

```
abstract class MenuAccel extends Menu {

  boolean processKey(KeyStroke ks) {
    for (int i = 0; i < itemCount(); i++) {
      if (((MenuItemAccel) itemAt(i)).processKey(ks)) return true;
    }
    return false;
  }

  void addAction(String label, Action action) {
    items.add(new MenuItemAccel(label, action));
  }
  …
}
```

Non-Explicit Covariant Dependencies

- Covariant dependencies between objects:
  - The varying functionality of an object in a group may need to access the corresponding varying functionality of another object of the group.
  - The type declarations in our design do not express covariant dependencies between the objects of a group.
  - References between objects are typed by invariant types, which provide a fixed interface.

```
abstract class MenuAccel extends Menu {

  boolean processKey(KeyStroke ks) {
    for (int i = 0; i < itemCount(); i++) {
      if (((MenuItemAccel) itemAt(i)).processKey(ks)) return true;
    }
    return false;
  }
  …
}
```

Covariant dependencies are emulated by type-casts.

Instantiation-Related Reusability Problems

- Code that instantiates the classes of an object group cannot be reused with different variations of the group.

```
abstract class Menu {

  void addAction(String label, Action action) {
    items.add(new MenuItem( // <= Creates a MenuItem
      label, action
    ));
  }
  …
}

abstract class MenuAccel extends Menu {

  void addAction(String label, Action action) {
    items.add(new MenuItemAccel( // <= Creates a MenuItemAccel
        label, action
    ));
  }
  …
}
```

Instantiation code can be spread all over the application.

Menu Contributor for Operations on Files

- A menu of an application can be built from different reusable pieces, provided by different menu contributors.

```
interface MenuContributor {
   void contribute(Menu menu);
}

class FileMenuContrib implements MenuContributor {
   void contribute(Menu menu) {
     CascadeMenuItem openWith = new CascadeMenuItem(”Open With”);
     menu.addItem(openWith);
     MenuItem openWithTE =
      new MenuItem(”Text Editor”, createOpenWithTEAction());
     openWith.addItem(openWithTE);

     MenuItem readOnly =
          new CheckMenuItem(”Read Only”, createReadOnlyAction());
     menu.addItem(readOnly)
     …
    }
    …
}
```

Instantiation-Related Reusability Problem

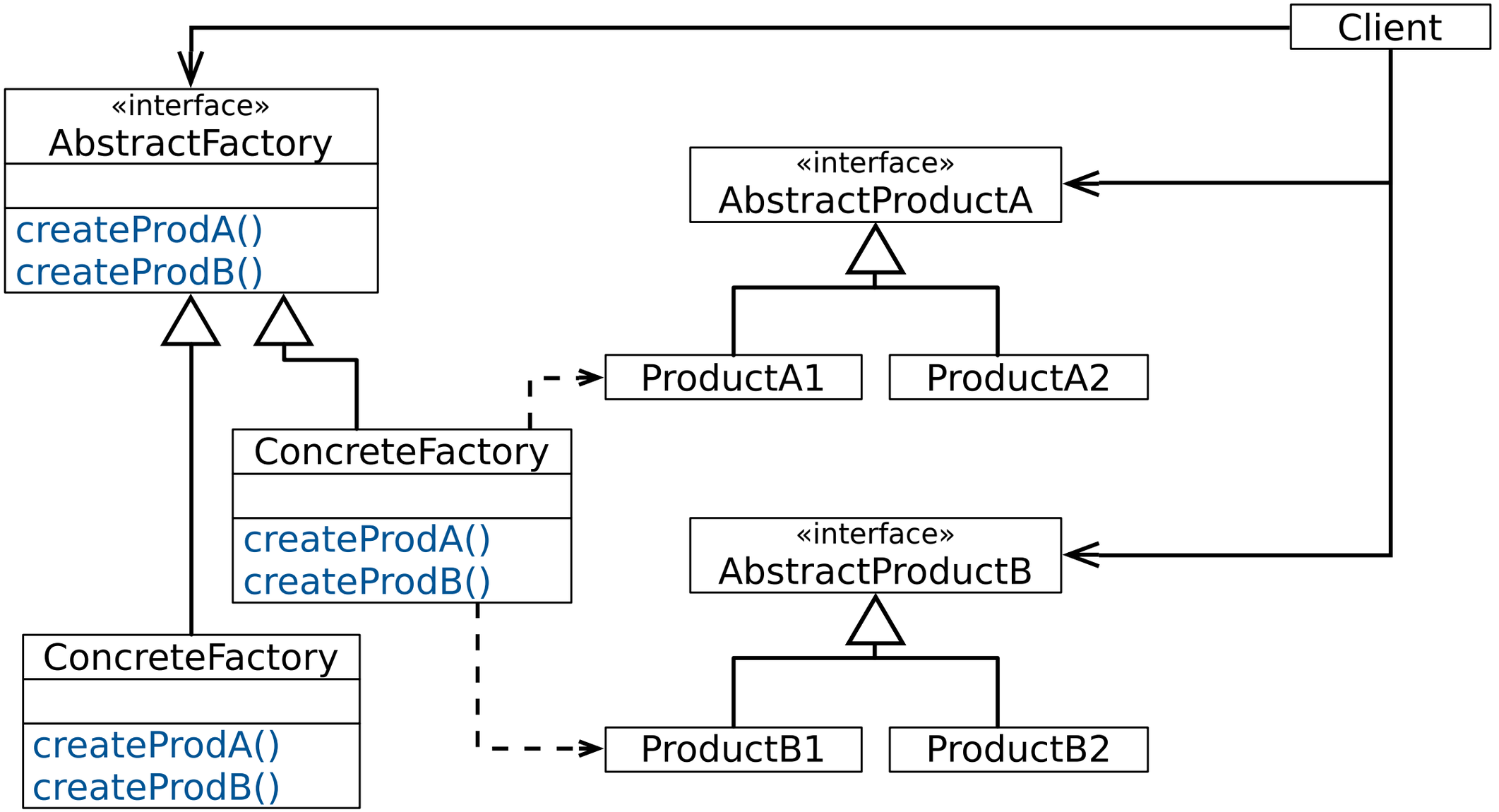
- In some situations, overriding of instantiation code can have a cascade effect.

- An extension of class C mandates extensions of all classes that instantiate C.

- This in turn mandates extensions of further classes that instantiate classes that instantiate C.

Can you imagine a workaround to address instantiation-related problems?

Abstract Factory Pattern



Factories for Instantiating Objects

```
interface MenuFactory {
  MenuItem createMenuItem(String name, Action action);
  CascadeMenuItem createCascadeMenuItem(String name);
  …
}
```

Factories for Instantiating Objects

```
class FileMenuContrib implements MenuContributor {

  void contribute(
      Menu menu,
      MenuFactory factory // <= we need a reference to the factory
  ) {
    MenuItem open = factory.createCascadeMenuItem(”Open”);
    menu.addItem(open);

    MenuItem openWithTE = factory.createMenuItem(...);
    open.addItem(openWithTE);

    …
    MenuItem readOnly = factory.createCheckMenuItem(...);
    menu.addItem(readOnly)

    …
  }

  …
}
```

Factories for Instantiating Objects

```
class BaseMenuFactory implements MenuFactory {
  MenuItem createMenuItem(String name, Action action) {
    return new MenuItem(name, action);
  }
  CascadeMenuItem createCascadeMenuItem(String name) {
    return new CasadeMenuItem(name);
  }
  …
}
```

```
class AccelMenuFactory implements MenuFactory {

  MenuItemAccel createMenuItem(String name, Action action) {
    return new MenuItemAccel(name, action);
  }

  CascadeMenuItemAccel createCascadeMenuItem(String name) {
    return new CasadeMenuItemAccel(name);
  }
  …
}
```

Deficiencies Of The Factory Pattern

- The infrastructure for the design pattern must be implemented and maintained.
- Increased complexity of design.
- Correct usage of the pattern cannot be enforced:
  - No guarantee that classes are instantiated exclusively over factory methods,
  - No guarantee that only objects are used together that are instantiated by the same factory.
- Issues with managing the reference to the abstract factory.
  - The factory can be implemented as a Singleton for convenient access to it within entire application.
    This solution would allow to use only one specific variant of the composite within the same application.
  - A more flexible solution requires explicit passing of the reference to the factory from object to object.
    Increased complexity of design.

Combining Composite & Individual Variation

Problem: How to combine variations of individual classes with those of features of a class composite.

- Feature variations at the level of object composites (e.g., accelerator key support).
- Variations of individual elements of the composite (e.g., variations of menus and items).

Menu Items with Accelerator Keys

```
class MenuItemAccel extends MenuItem {

    KeyStroke accelKey;
    boolean processKey(KeyStroke ks) {
      if (accelKey != null && accelKey.equals(ks)) {
        performAction();
        return true;
      }
      return false;
    }
    void setAccelerator(KeyStroke ks) { accelKey = ks; }
    void draw(Graphics g) { super.draw(g); displayAccelKey(); }
    …

}

class CascadeMenuItemAccel extends ???
class CheckMenuItemAccel extends ???
class RadioMenuItemAccel extends ???
```

How to extend subclasses of `MenuItem` for different variants of items with the accelerator key feature?
We need subclasses of them that also inherit the additional functionality in `MenuItemAccel`.

Menus with Accelerator Keys

```
abstract class MenuAccel extends Menu {

  boolean processKey(KeyStroke ks) {
    for (int i = 0; i < itemCount(); i++) {
      if (((MenuItemAccel) itemAt(i)).processKey(ks)) return true;
    }
    return false;
  }

  void addAction(String label, Action action) {
    items.add(new MenuItemAccel(label, action));
  }
  …

}

class PopupMenuAccel extends ???
class MenuBarAccel extends ???
```

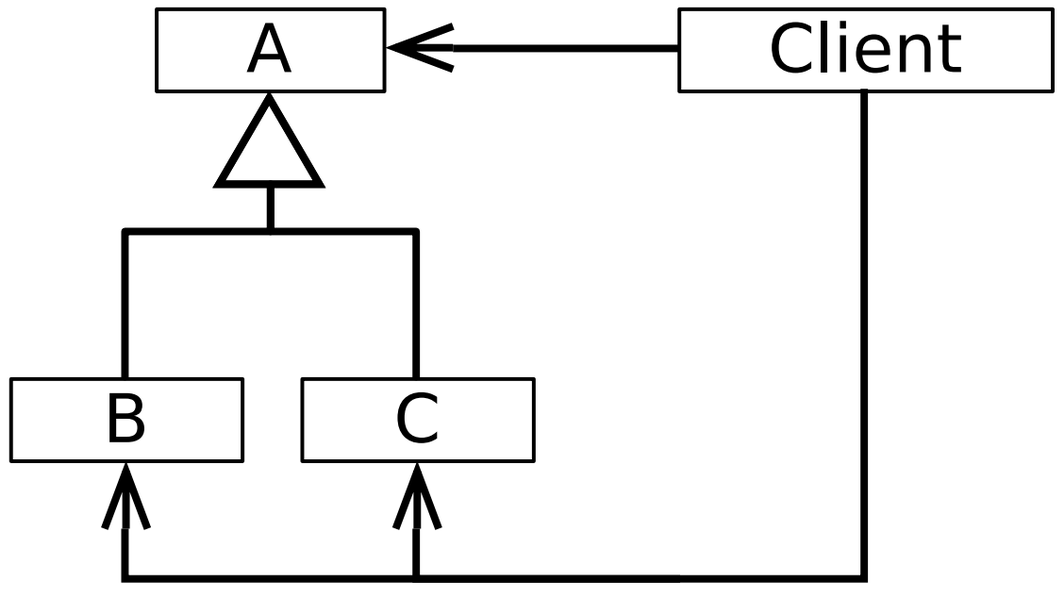
How to extend subclasses of Menu with the accelerator key feature?
We need subclasses of them that also inherit the additional functionality in MenuAccel.

In languages with single inheritance, such as Java, combining composite & individual variations is non-trivial and leads to code duplication.

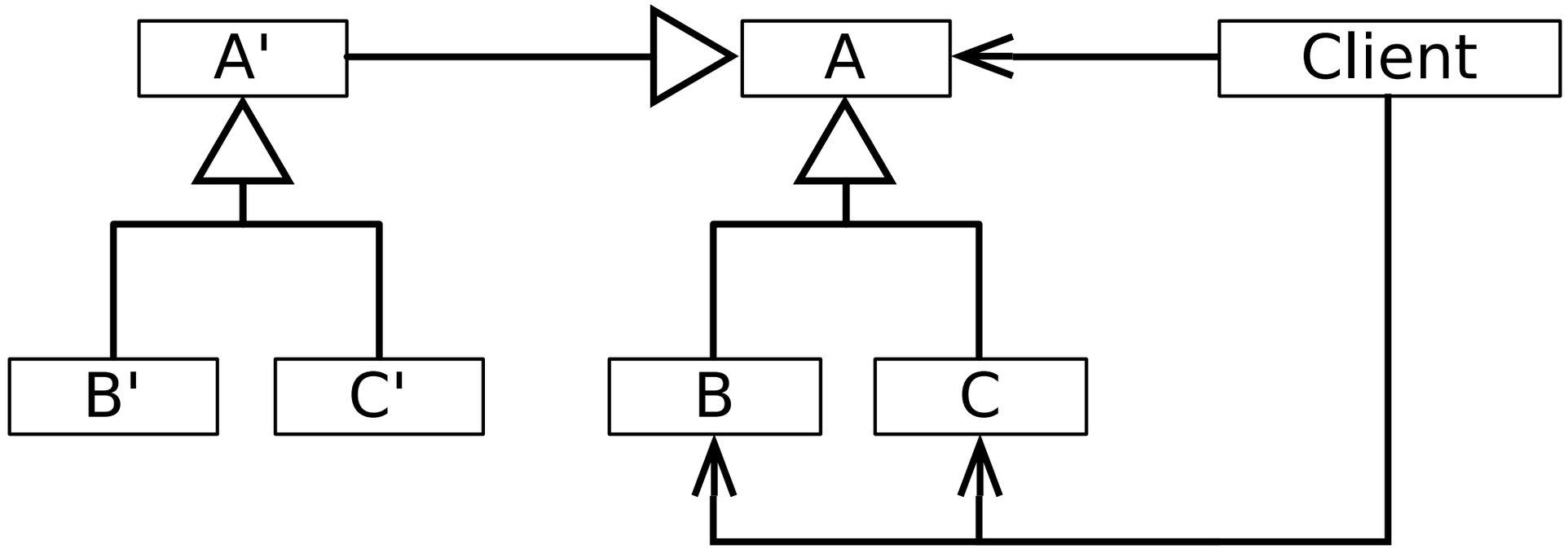
The Problem in a Nutshell

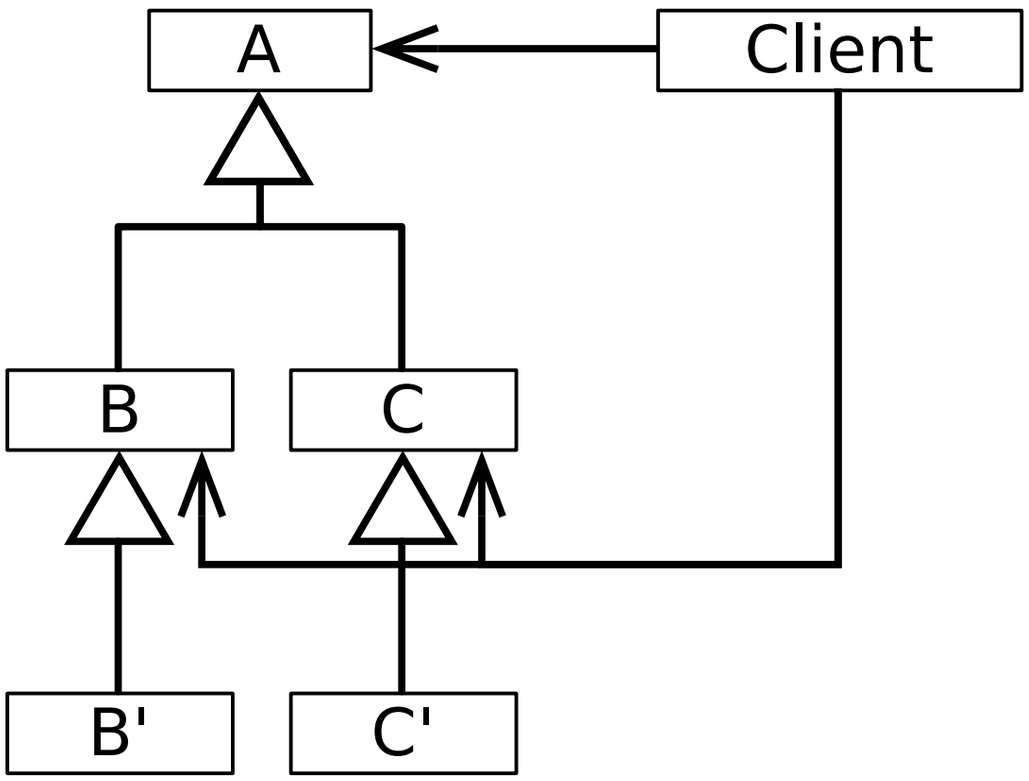
- We need to extend A (and in parallel to it also its subclasses B and C) with an optional feature (**should not necessarily be visible to existing clients**).

- This excludes the option of modifying A in-place, which would be bad anyway because of OCP.



Alternative Designs





Combining Composite and Individual Variations

Using some form of multiple inheritance…

```
class PopupMenuAccel extends PopupMenu, MenuAccel { }
class MenuBarAccel extends MenuBar, MenuAccel { }
```
```

```Java
class CascadeMenuItemAccel extends CascadeMenuItem, MenuItemAccel {
  boolean processKey(KeyStroke ks) {
    if (((PopupMenuAccel) menu).processKey(ks) ) return true;
    return super.processKey(ks);
  }
}

class CheckMenuItemAccel extends CheckMenuItem, MenuItemAccel { ... }
class RadioMenuItemAccel extends RadioMenuItem, MenuItemAccel { ... }
```
```

Summary

- General agreement in the early days of OO:
  Classes are the primary unit of organization.
  - Standard inheritance operates on isolated classes.
    - Variations of a group of classes can be expressed by applying inheritance to each class from the group separately.
- Over the years, it turned out that sets of collaborating classes are also units of organization. In general, extensions will generally affect a set of related classes.

(Single-) Inheritance **does not appropriately support OCP with respect to changes that affect a set of related classes**!

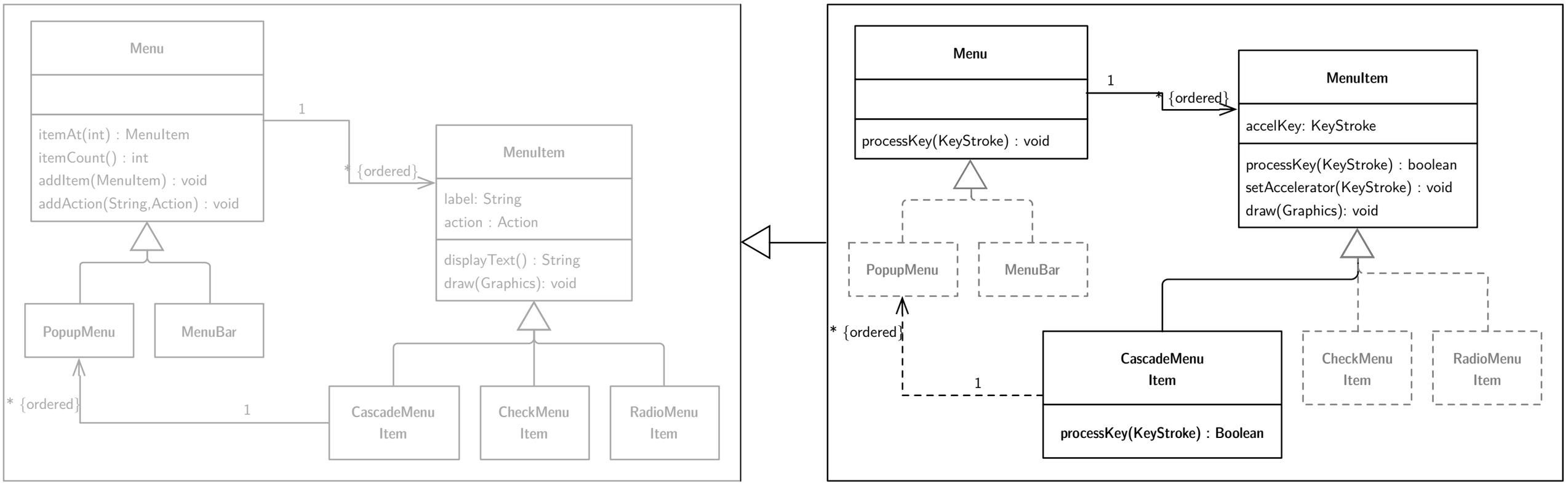
Almost all features that proved useful for single classes are not available for sets of related

Desired/Required Features

- **Incremental programming at the level of sets of related classes.**
  In analogy to incremental programming at the level of individual classes enabled by inheritance.
  (I.e., we want to be able to model the accelerator key feature by the difference to the default menu functionality.)

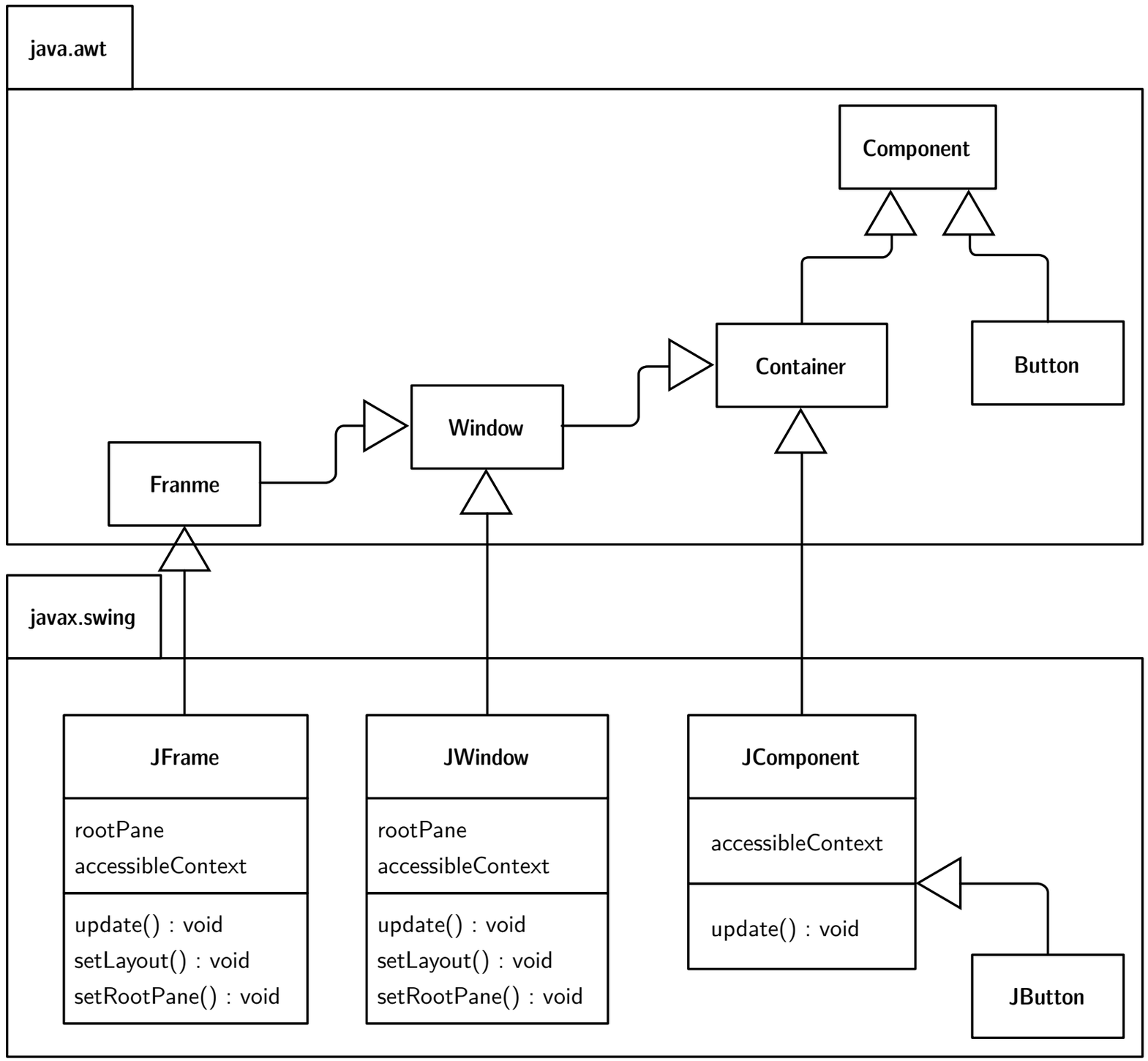
- **Polymorphism at the level of sets of related classes → Family polymorphism.**
  In analogy to subtype polymorphism at the level of individual classes.
  (I.e., we want to be able to define behavior that is polymorphic with respect to the particular object group variation.)

Family Polymorphism



The Design of AWT and Swing

A small subset of the core of AWT (Component, Container, Frame, Window) and Swing.



AWT Code

```
public class Container extends Component {
  int ncomponents;
  Component components[] = new Component[0];

  public Component add (Component comp) {
    addImpl(comp, null, -1);
    return comp;
  }

  protected void addImpl(Component comp, Object o, int ind) {
    …
    component[ncomponents++] = comp;
    …
  }

  public Component getComponent(int index) {
    return component[index];
  }
}
```

The code contains no type checks and/or type casts.

Swing Code

The code contains type checks and/or type casts.

```
public class JComponent extends Container {

  public void paintChildren (Graphics g) {
    …
    for (; i > = 0 ; i--) {
      Component comp = getComponent (i);
      isJComponent = (comp instanceof JComponent); // type check
      …
      ((JComponent)comp).getBounds(); // type cast
      …
    }
  }
}
```

About the Development of Swing

"In the absence of a large existing base of clients of AWT, Swing might have been designed differently, with AWT being refactored and redesigned along the way.

Such a refactoring, however, was not an option and we can witness various anomalies in Swing, such as duplicated code, sub-optimal inheritance relationships, and excessive use of run-time type discrimination and downcasts."

Takeaway

- Inheritance is a powerful mechanism for supporting variations and stable designs in presence of change. Three desired properties:
  - **Built-in support for OCP** and reduced need for preplanning and abstraction building.
  - **Well-modularized** implementations of variations.
  - **Support for variation of structure/interface** in addition to variations of behavior.
  - **Variations** can participate in **type declarations**.

Takeaway

- Inheritance has also deficiencies
  - Variation implementations are not reusable and not easy to compose.
    - Code duplication.
    - Exponential growth of the number of classes; complex designs.
- Inheritance does not support dynamic variations – configuring the behavior and structure of an object at runtime.
- Fragility of designs due to lack of encapsulation between parents and heirs in an inheritance hierarchy.
- Variations that affect a set of related classes are not well supported.