Summer Term 2018

Software Engineering Design & Construction

Dr. Michael Eichberg
Fachgebiet Softwaretechnik
Technische Universität Darmstadt

A Critical View on Inheritance

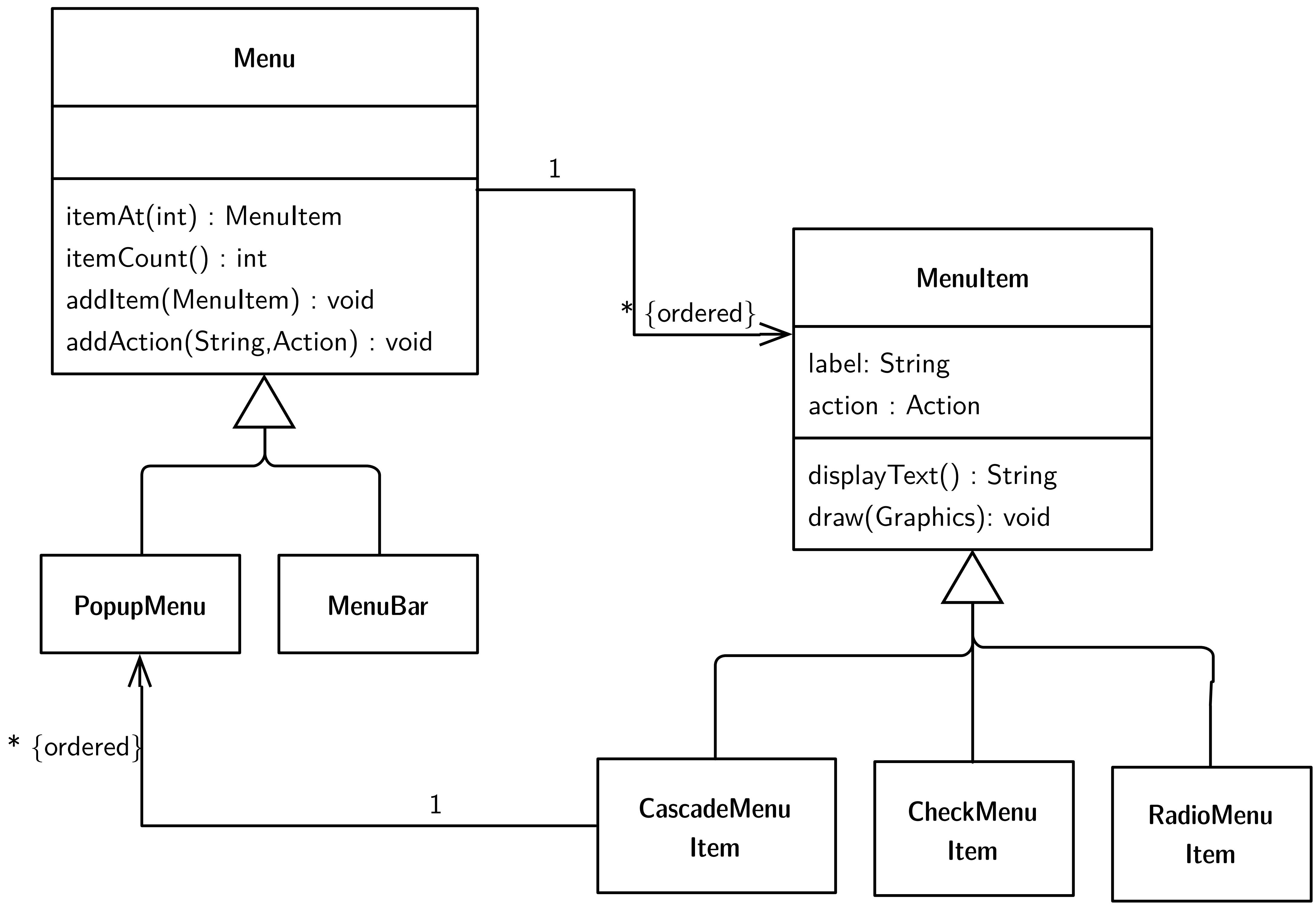
Variations at the Level of Multiple Objects

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.

```java
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.

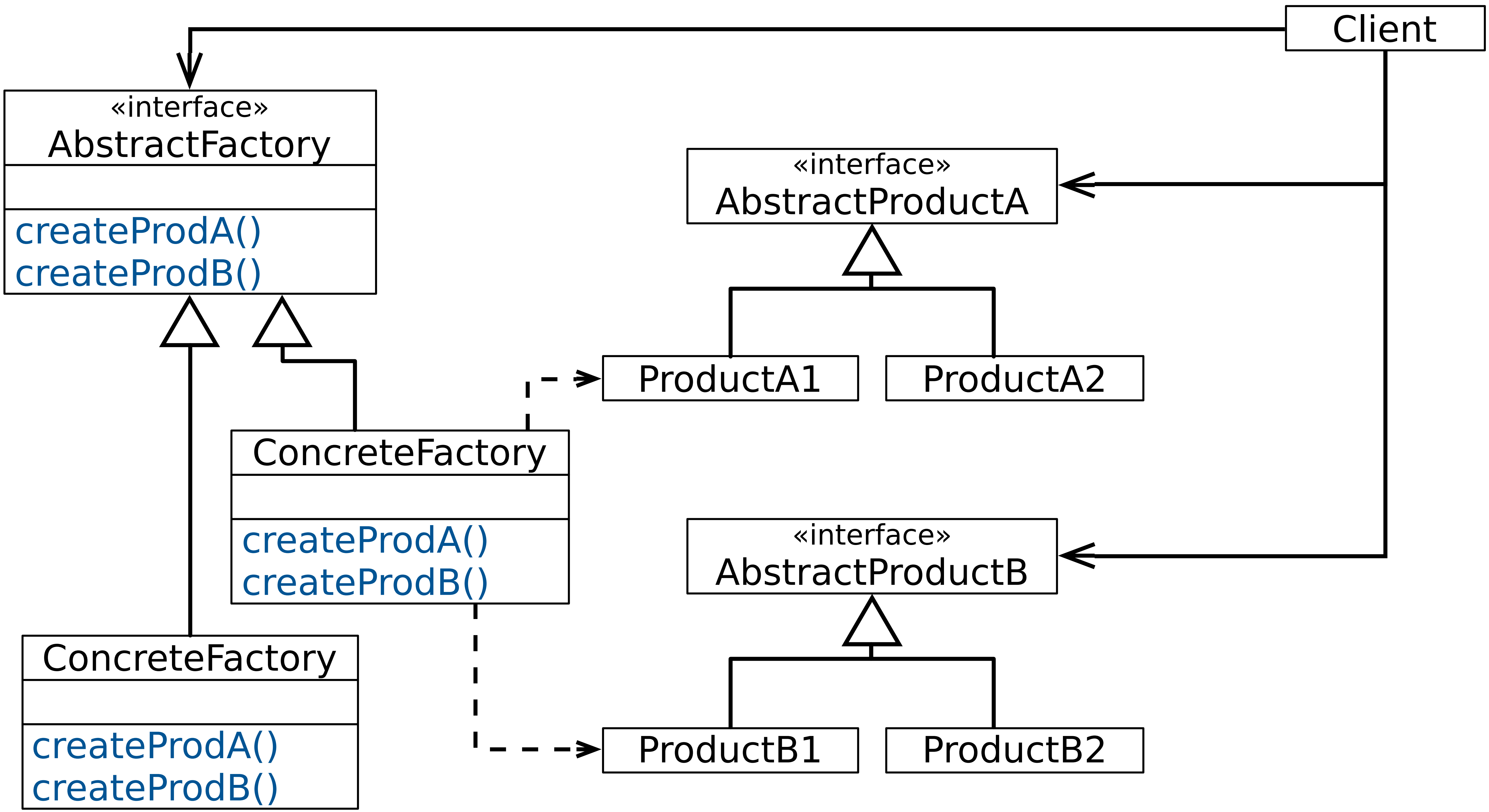
```
interface MenuContributor {
   void contribute(Menu menu);
}

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 Abstract 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.

Combining Composite & Individual Variation

Problem: How to combine variations of individual classes with those 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 ???
```

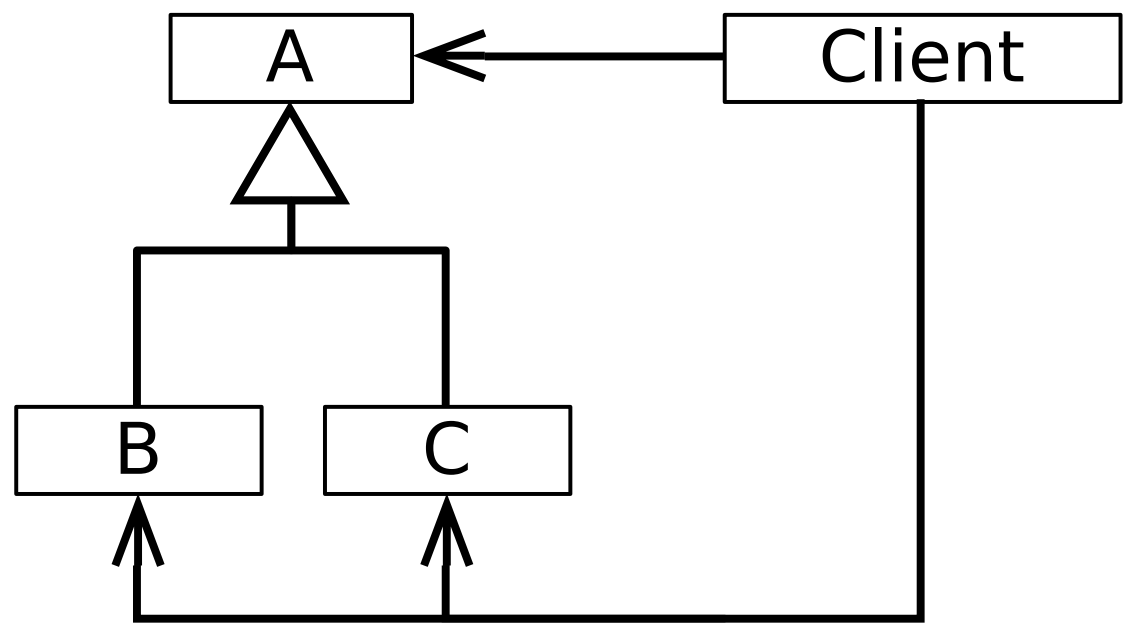
Menus with Accelerator Keys

```
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 ???
```

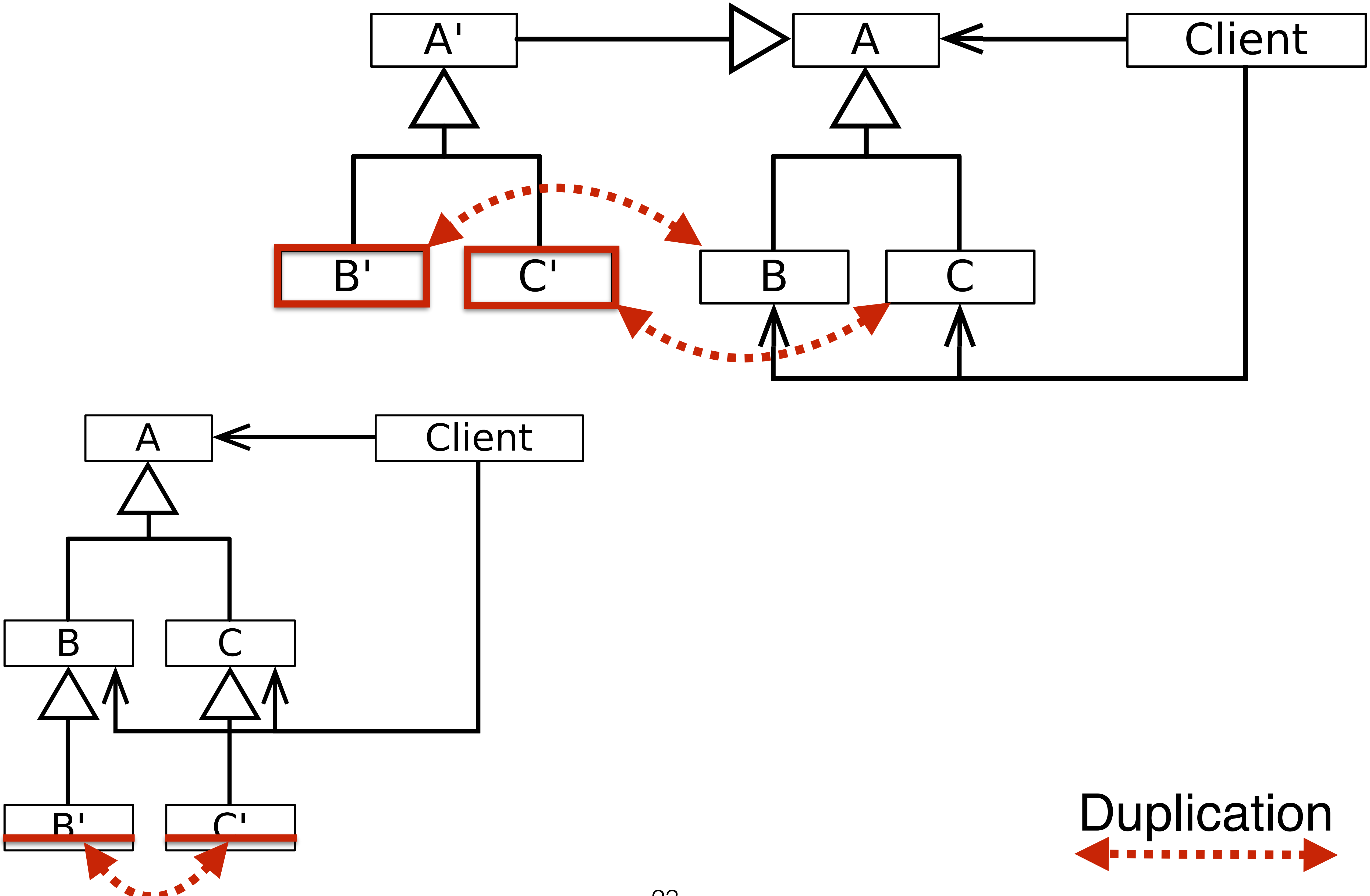
In languages with single inheritance, such as Java, combining composite & individual variations is non-trivial and leads to code duplication.

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 { }

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**!

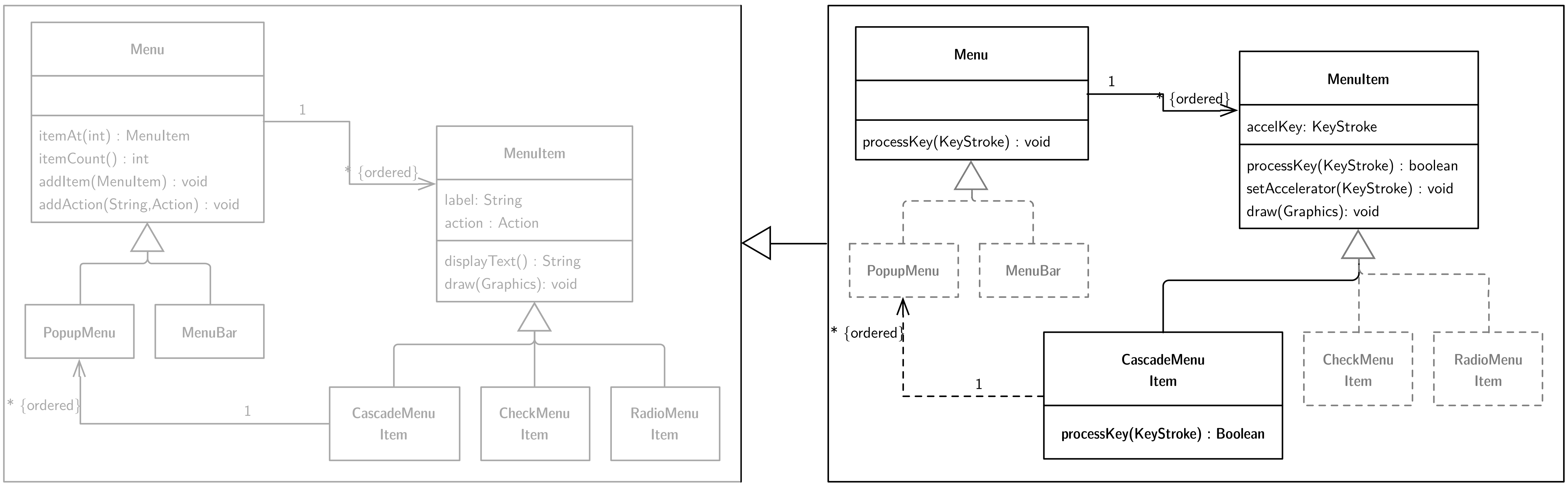
Almost all features that proved useful for single classes are not available for sets of related

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.)

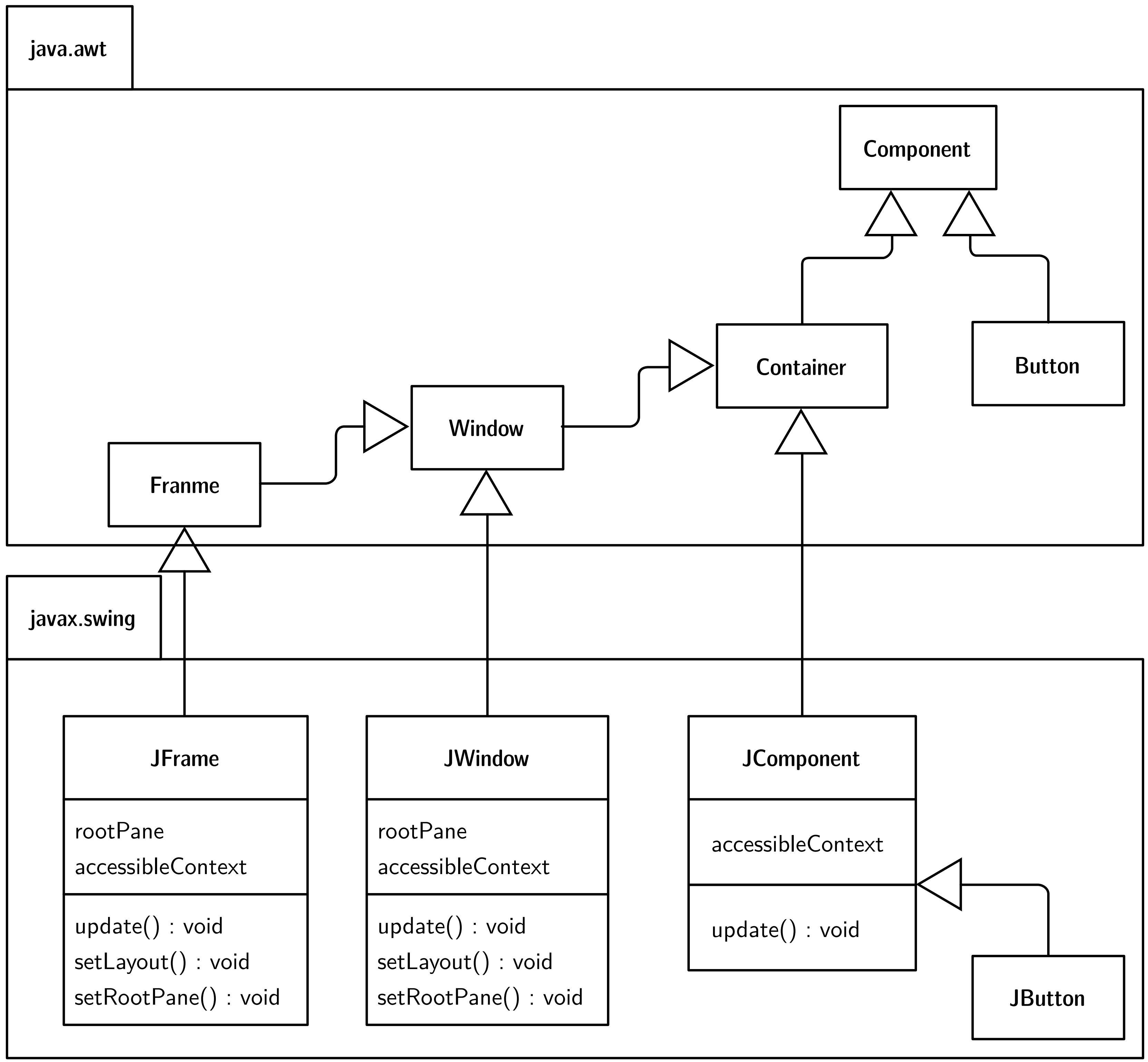
Family Polymorphism



Conceptual View
This is not valid UML.

The Design of AWT and Swing

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



AWT Code

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

```
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];
  }
}
```

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.