Summer Term 2018

Software Engineering Design & Construction

Dr. Michael Eichberg
Fachgebiet Softwaretechnik
Technische Universität Darmstadt

A Critical View on Inheritance

A smart home has many features that are controlled automatically:
Heating, Lighting, Shutters, …

We want to develop a software that helps us to control our smart home.

Variations at the Level of Multiple Objects

So far, we considered variations, whose scope are individual classes.
But, no class is an island!

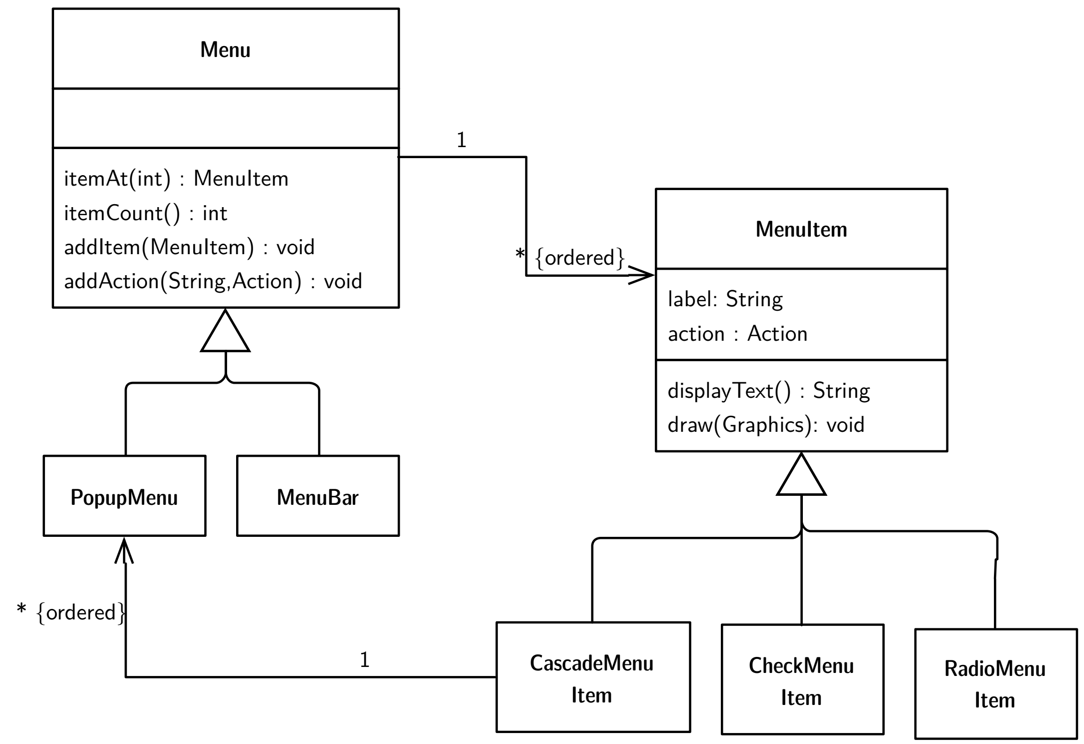
Examples of class groupings:

- data structures such as trees and graphs,
- sophisticated frameworks,
- the entire application.

Classes in a group may be related in different ways:
- by references to each other,
- by signatures of methods and fields,
- by instantiation,
- by inheritance,
- by shared state and dependencies.

Window Menus

Illustrative Example



For illustration, we will consider variations of menu structures:

- A menu is a GUI component consisting of a list of menu items corresponding to different application-specific actions.
- Menus are usually organized hierarchically: a menu has several menu items.
- There may be different variants of menus (popup, menu bar).
- There may be different variants of menu items.
- A menu item can be associated with a cascade menu which pops up when the item is selected.

Menu and menu item objects are implemented by *multiple classes that are organized in inheritance hierarchies* to represent variations of the elements of the object structure.

- A menu represented by class Menu maintains a list of menu items.
- Subclasses of Menu implement specialized menus.
- A PopupMenu is a subclass of Menu implementing pop-up menus.
- MenuBar is a subclass of Menu, implementing a menu bar which is usually attached at the top edge of a window and serves as the top level menu object of the window.
- Simple menu items are implemented by class MenuItem
- Subclasses of MenuItem implement specialized menu items:
    - class CheckMenuItem for check-box menu items,
    - class RadioMenuItem for radio-button menu items,
    - CascadeMenuItem for menu items that open cascade menus. It contains a reference to an instance of a PopupMenu, a subclass of Menu implementing pop-up menus.

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 { ... }
```


Classes involved in the implementation of menu functionality refer to each other in the declarations and implementations of their fields and methods.

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*.

Various optional features related to functionality of menus:

- Support for accelerator keys for a quick selection of a menu item using a specific key stroke,
- Support for multi-lingual text in menu items,
- Support for context help.

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();
  }
  …
}
```


The extension of menu items with accelerator keys is implemented in class MenuItemAccel, a subclass of MenuItem.

The extension affects both: the implementation of existing methods as well as the structure and interface of menu items. E.g., the implementation of the draw method needs to be extended to display the accelerator key besides the label of the item.

New attributes and methods are introduced:

- to store the key associated to the menu item,
- to change this association,
- to process an input key,
- to display the accelerator key.

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));
  }
  …
}
```


MenuAccel implements the extension of menus with accelerator keys:

- adds the new method processKey for processing keys
- overrides method addAction to ensure that the new item added for an action supports accelerator keys

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.

The method processKey in a menu with accelerator keys needs to call processKey on its items.

- Items of a menu are accessed by calling the method itemAt.
- The method itemAt is inherited from class Menu, where it was declared with return type MenuItem.
- Thus, to access the extended functionality of menu items, we must cast the result of `itemAt` to `MenuItemAccel`.

The design cannot guarantee that such a type cast will always be successful, because items of MenuAccel are added over the inherited method addItem, which accepts all menu items, both with and without the accelerator functionality.

Potential for LSP violation!

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.

- MenuItem is instantiated in Menu.addAction(...).
- In MenuAccel, we override addAction(...), so that it instantiates MenuItemAccel.

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

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)
    …
   }
   …
}
```

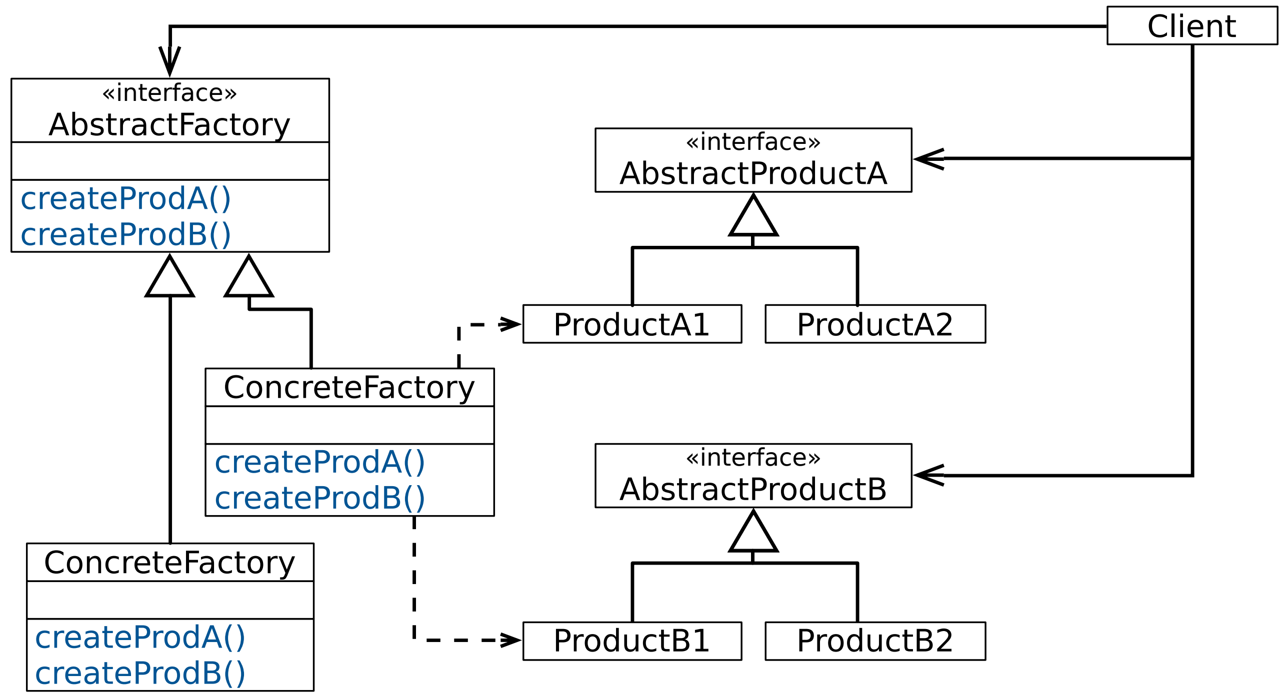

The code shows the implementation of a menu contributor for operations on files. It implements the method contribute, which extends the given menu object with menu items to open files with different text editors, to change the read-only flag of the file, and so on. Since the menu items are created by directly instantiating the respective classes, this piece of code cannot be reused for menus with support for key accelerators or any other extensions of the menu functionality.

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



The described problem and its solution is so common, that a well-known pattern (Abstract Factory) exists which helps you to solve it!

(We will discuss this and other patterns in more detail later!)

Factories for Instantiating Objects

```
interface MenuFactory {

  MenuItem createMenuItem(String name, Action action);

  CascadeMenuItem createCascadeMenuItem(String name);

  …
}
```


The **Abstract Factory design pattern** enables abstraction from group variations by late-bound instantiation of the classes of the group's objects.

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.

Several studies have shown that the comprehensibility of some code/framework significantly decreases, when it is no longer possible to directly instantiate objects.

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 ???
```


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?

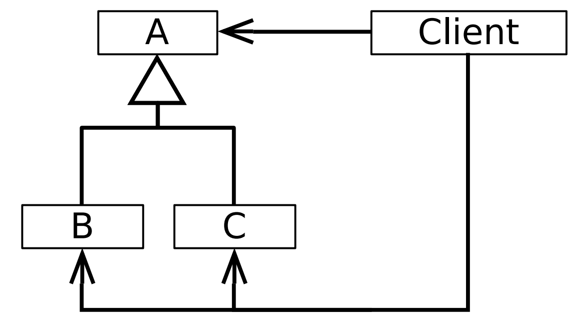
We need subclasses of them that also inherit the additional functionality in `MenuAccel`.

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

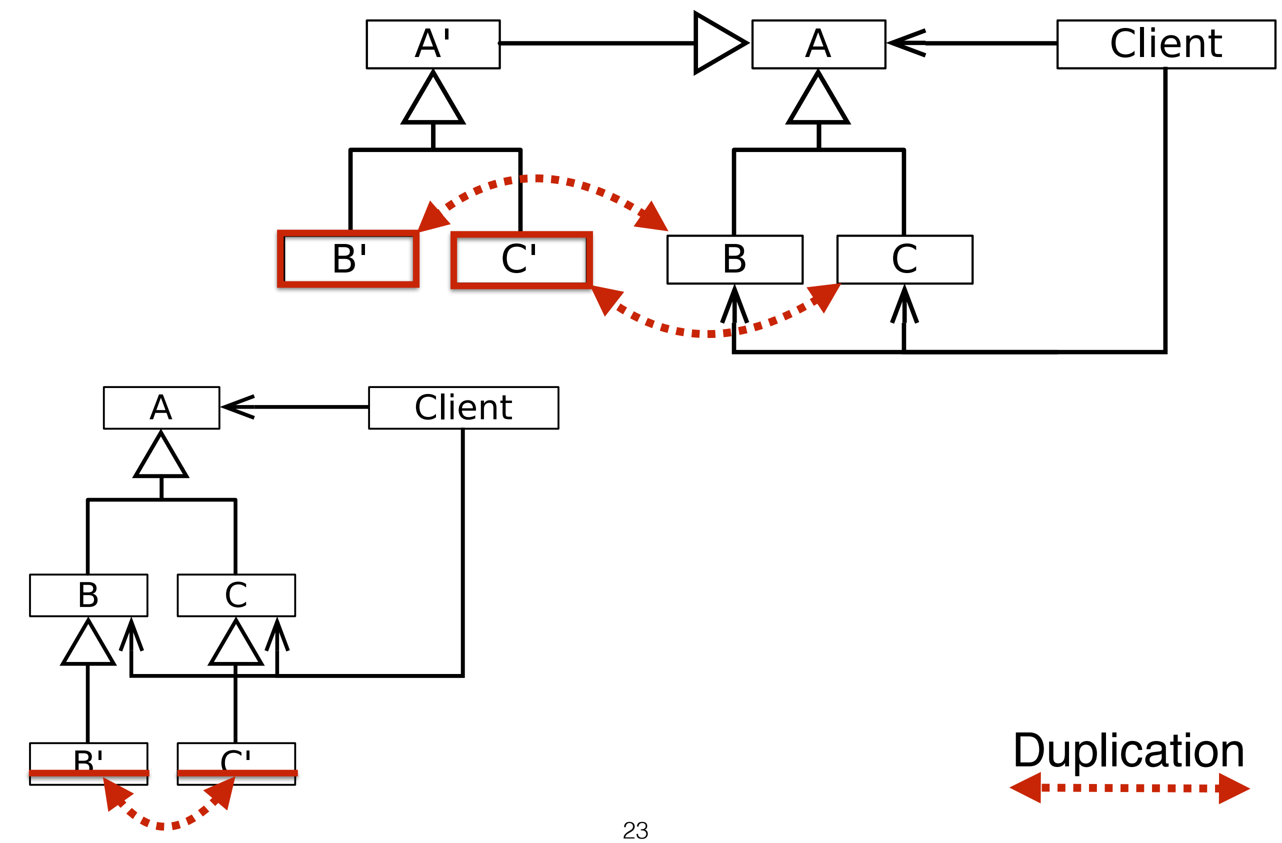
The Problem in a Nutshell

- 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



There are two possibilities:

1. creating a parallel hierarchy or
2. creating additional subclasses of B and C) to add an optional feature to A incrementally without affecting clients in a *single inheritance setting.*

In both cases, code needs to be duplicated which leads to a maintenance problem.
(In the first case B and C need to be duplicated.)
(In the second case the (new) method needs to be added to B’ and C’ - along with the required fields!)

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 { ... }
```


The design with multiple inheritance has its own problems:

It requires explicit, additional class declarations that explicitly combine the extended element class representing the composite variation with sub-classes that describe its individual variations.

- Such a design produces an **excessive number of classes**.
- The design is also **not stable with respect to extensions** with new element types.
- The developer must not forget to extend the existing variations of the composite with combinations for the new element types.

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

Mainstream OO languages (incl. Scala!) have only insufficient means for organizing collaborating classes: packages, name spaces, etc. These structures have serious problems:

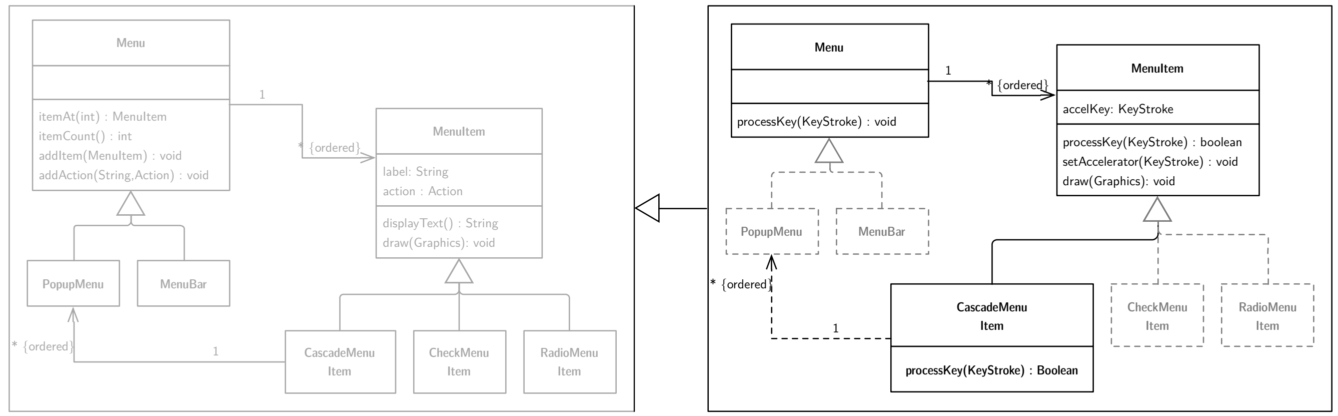
- No means to express variants of a collaboration.
- No polymorphism.
- No runtime semantics.

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.

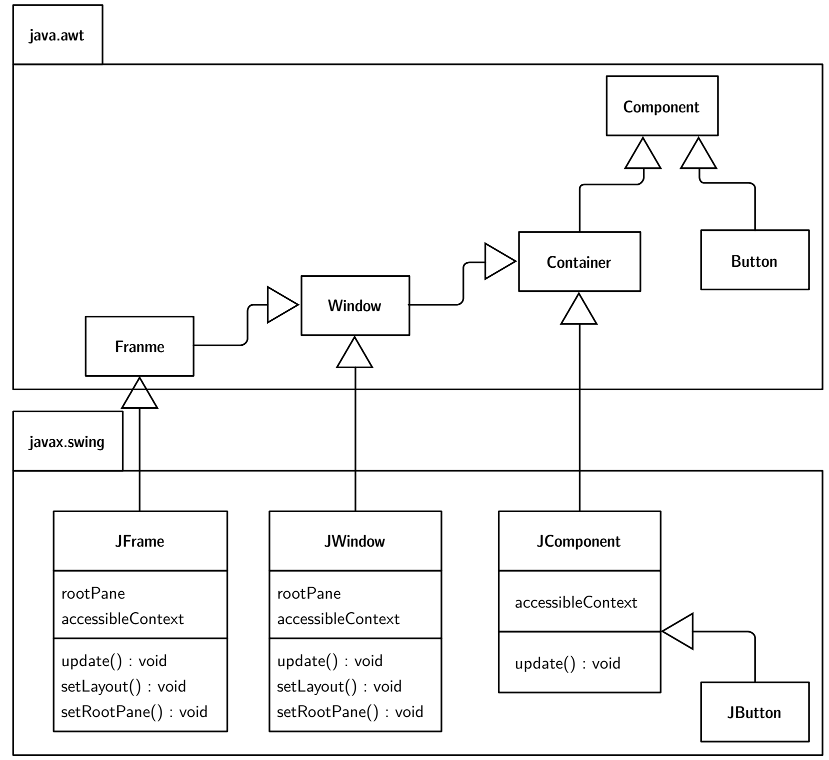
We want to avoid:

- code duplication
- casts
- the necessity to re-implement methods (e.g. addAction)

Ideally would like to have several versions of class definitions - one per responsibility - which can be mixed and matched on-demand.

The Design of AWT and Swing

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



The question may arise whether this is this a real problem (modification of family of related classes) or not. As we will see in the following it is a very real problem which even shows up in mature deployed software.

Case Study: Java AWT and Swing

Some of the material used in the following originally appeared in the paper: *Bergel et al, Controlling the Scope of Change in Java, International Conference on Object-Oriented Programming Systems Languages and Applications 2005*

AWT is a GUI framework that was included in the first Java release and which directly interfaces the underlying operating system. Therefore, only a small number of widgets are supported to make code easier to port.

Swing extends AWT core classes (by subclassing) with functionality such as: "pluggable look and feel" and "double buffering". The Swing-specific support for double buffering to provide smooth flicker-free animation is implemented, among others, in the methods update(), setLayout(), etc.. Furthermore, Swing adds more widgets.

Issues:

- Features defined in JWindow are duplicated in JFrame. Due to the absence of an inheritance link between JFrame and JWindow (JWindow: 551 LOC; JFrame: 829 LOC, 241 lines of code are duplicated; 43% of JWindow reappears as 29% of JFrame.
- While a Window is a Component in AWT, a JWindow is not a JComponent in Swing.
- While a Button is a Component and JButton is a JComponent, a JButton is not a Button!
- A Swing Component is a Container for other components.
- Feature inherited from Container (JComponent extends Container).
- Types of subcomponents in Container are Component not JComponent.
- Ubiquitous runtime type checks and type casts are the result!

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.