Software Design & Programming Techniques

A Critical View on Inheritance

Prof. Dr-Ing. Klaus Ostermann

Based on slides by Prof. Dr. Mira Mezini

Goals of this Lecture Block

- Inheritance is the main built-in variability mechanism of OO languages.
- Common functionality of similar objects can be implemented by a base class and each variation can be implemented by a separate subclass.

- In this block, we analyze its advantages and disadvantages with respect to support for variability.
- Many design patterns propose solutions to compensate for deficiencies of inheritance.
- This lecture is a bridge between design principles and the lecture blocks about design patterns and advanced languages.

A Critical View on Inheritance

4.1 Desired Properties of Inheritance

Desired Properties in a Nutshell

Built-in support for OCP reduces the need to anticipate variations.

- Inheritance allows replacing the implementation of arbitrary methods of a base class.

  [unless forbidden, e.g., in Java methods can be declared as `final`]
- No need to plan in advance, which methods will vary, and to design appropriate abstractions for that purpose.
- Of course, support for variability in a class is conditioned by the granularity of its methods.

Good modularity.

- The base class is free of any variation-specific functionality.
- Each variation is implemented in a separate subclass.

Desired Properties in a Nutshell

Support for structural variations.

- Subclasses can vary the implementation of the inherited base interface.
- In addition, inheritance allows to design the most suitable interface for each variation.
- Different variations of an object type may need to extend the base interface with variation-specific fields and methods.

Variations can be represented in type declarations.

This is necessary for type-safe access of variation-specific interfaces.

Desired Properties by Example

- Let us consider variations of selection functionality of table widgets.
- The modularization of these variations by inheritance is illustrated by the (pseudo-)code in the following slide.

The Table Selection Hierarchy

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

The Table Selection Hierarchy

- `TableBase` implements basic functionality of table widgets as a variation of the functionality common for all widgets, e.g., display of tabular data models.
- The abstract class `TableSel` extends `TableBase` with functionality that is common for all types of table selection, e.g., rendering of selected cells.
- Finally, `TableSingleCellSel`, `TableSingleRowSel`, `TableRowRangeSel`, and `TableCellRangeSel` implement specific types of table selections.

Desired Properties by Example

- **Built-in support for OCP**

  The implementation of method `paintCell` in `TableSel` can be changed without pre-planning.

- **Good modularity**

  Each table selection model is encapsulated in a separate class.

- **Support for structural variations**
  - Different operations and variables are declared and implemented by `TableSingleCellSel` and `TableSingleRowSel`: `currRow`, `currCel`, `selectCell` and `currRow`, `selectRow`, respectively.
  - Can design the most suitable interface for each type of table selection.
  - Do not need to design a base interface that fits all future variations.

- **Variations can be represented in type declarations**

  We know that a variable declared with type `TableSingleRowSel` would always refer to a table supporting single row selection.

4.2 Deficiencies of Inheritance

4.2 Deficiencies of Inheritance

- 4.2.1 Non-Reusable, Hard-to-Compose Extensions
- 4.2.2 Weak Support for Dynamic Variability
- 4.2.3 The Fragile Base Class Problem
- 4.2.4 Variations at the Level of Multiple Objects

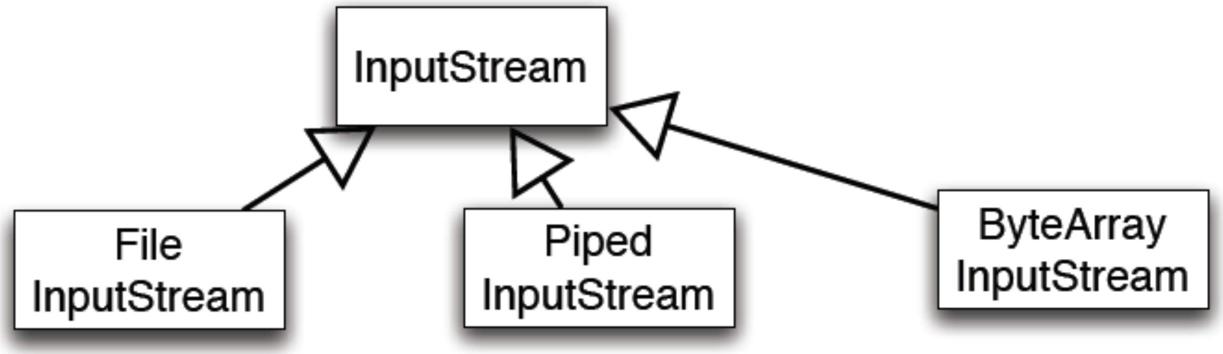
4.2.1 Non-Reusable, Hard-to-Compose Extensions

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

An Extract from Java’s Stream Hierarchy

- Consider an extract from `java.IO` package.
- Consisting of classes for reading from a source.

- Streams abstract from concrete data sources and sinks.



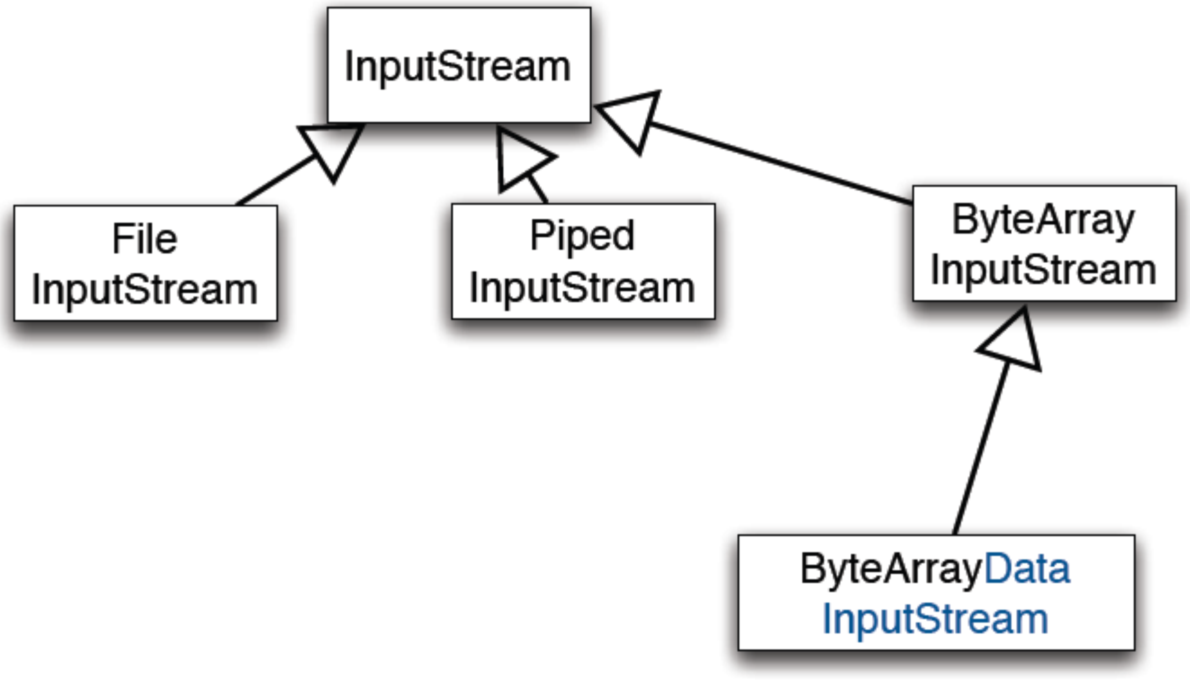
An Extract from Java’s Stream Hierarchy

- `InputStream` is root of stream classes reading from a data source.
- `FileInputStream` implements streams that read from a file.
- `PipedInputStream` implements streams that read from a `PipedOutputStream`.

  Typically, a thread reads from a `PipedInputStream` data written to the corresponding `PipedOutputStream` by another thread.
- `ByteArrayInputStream` implements streams that read from memory.

A Simple Variation

- Need a variation of `ByteArrayInputStream` capable of reading whole sentences and not just single bytes.
- Can implement it as a subclass of `ByteArrayInputStream`.
- "Blue" in the name of the class denotes the delta needed to implement this variation.



Further Variations

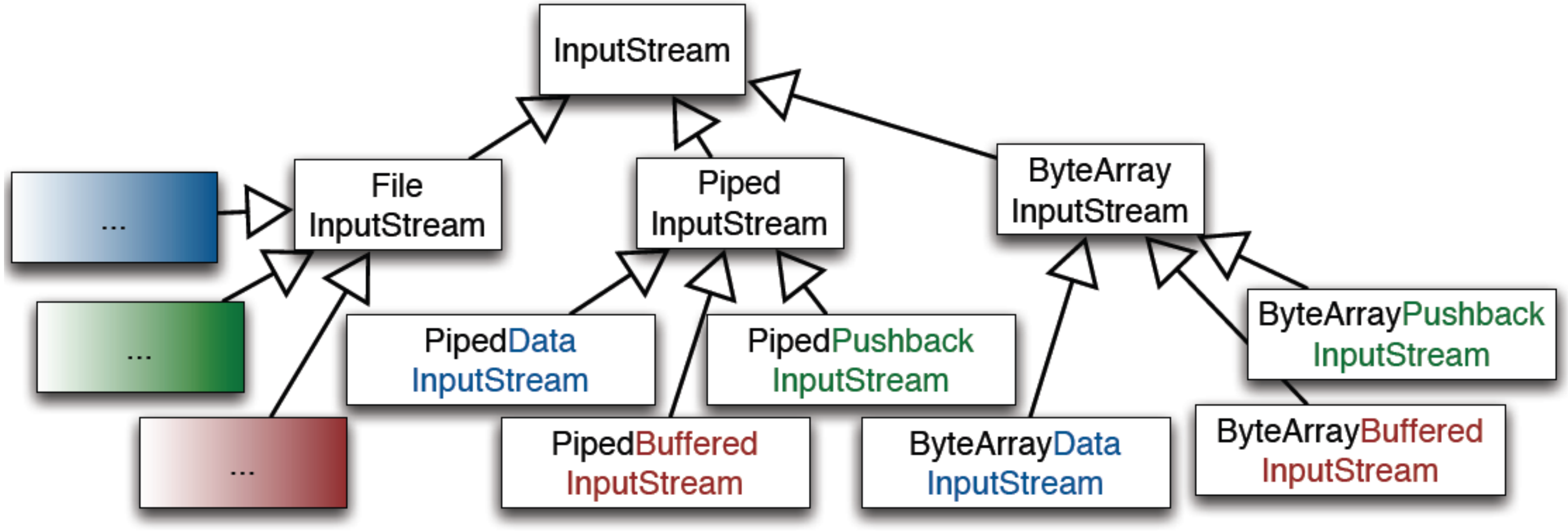
- **Reading whole sentences with other kinds of streams**:
  - `FileInputStream` objects that re able to read whole sentences.
  - `PipedInputStream` should read whole sentences too.
  - …

- **Other kinds of variations** on stream functionality
  - Writing the given data back (“red” in the following slide)
  - Buffering content (“green” in the following slide),
  - Counting the numbers of lines processed,
  - …

Non-Reusable and Hard to Combine Stream Variants

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

- Huge amount of code duplication.
- Complex design.


InputStream
File InputStream
Piped InputStream
ByteArray InputStream
...
...
...
PipedData InputStream
PipedPushback InputStream
PipedBuffered InputStream
ByteArrayData InputStream
ByteArrayPushback InputStream
ByteArrayBuffered InputStream

Non-Reusable, Hard-to-Compose Extensions

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

- A particular type of variation need to be re-implemented for all siblings of a base type.
- **Code duplication** is the result.

- Large number of independent extensions are possible:
  - For every new functionality we want.
  - For every combination of every functionality we want.

- **Maintenance nightmare**: exponential growth of number of classes.

- This problem was the motivation for many proposed extensions to OO programming
  - Mixins, traits

4.2.2 Weak Support for Dynamic Variability

- The configuration of an object' implementation may depend on values from the runtime context.
- Example:
  - Table widget options may come from some dynamic configuration panel;
  - Depending on the configuration options different compositions of table widget features need to be instantiated.

- Mapping from runtime values to classes to be instantiated can be implemented by conditional statements.
- 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.

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

4.2.3 The Fragile Base Class Problem

Changes in base class may lead to unforeseen problems in subclasses. **“Inheritance Breaks Encapsulation”**

*) Slides in this section discuss material from Item 17 of Joshua Bloch, Effective Java

4.2.3.1 The Fragile Base Class in a Nutshell

- You can modify a base class in a seemingly safe way.
- But this modification, when inherited by the derived classes, might cause them to malfunction.

- **You can't tell whether a base class change is safe simply by examining the base class' methods in isolation.**
  - You must look at (and test) all derived classes as well.
  - Moreover, you must check all code that uses both the base class and its derived classes, since this code might also be broken by the changed behavior.

- **A simple change to a key base class can render an entire program inoperable.**

4.2.3.2 The Fragile Base Class Illustrated

- Suppose we want to implement `HashSet`s that know the number of their elements.

- We implement a class `InstrumentedHashSet` that inherits from `HashSet` and overrides methods that change the state of a `HashSet` …

An Instrumented HashSet

```
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());
    }
}
```

Question:
What is printed on the screen?

An Instrumented HashSet

```
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(Collect
        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());
    }
}
```

Question:
What is printed on the screen?

The answer is 6.

The reason:
The implementation of `addAll` in `HashSet` internally makes a this-calls on `add`.
Hence, added elements are double counted.

An Instrumented HashSet

```
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());
    }
}
```

Will the problem be solved if we do not override `addAll`?

An Instrumented HashSet

```
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(Coll
        addCount += c.size();
        return super.addAll(c);
    }
    public int getAddCount() {
        return addCount;
    }

    public static void main(String[] ar
        InstrumentedHashSet<String> s =
        s.addAll(Arrays.asList("aaa", `
        System.out.println(s.getAddCour
    }
}
```

Will the problem be solved if we do not override `addAll`?

For the moment yes. But not in general.

What if in the future the designers of `HashSet` decide to re-implement `addAll` to insert the elements of the parameter collection as a block rather than by calling `add` on each element of the collection? Might be necessary for efficiency reasons.

4.2.3.3 Fragile Base Class Continued

- The kind of fragility considered so far is caused by changes in 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 behavior is broken.

Adding New Methods May Break Subclasses

Another kind fragility is caused by extensions of the base class with new methods that were not there when the class was subclassed.

Adding New Methods May Break Subclasses

Fragility caused by extensions of the base class.

Example:

- Consider a base collection class.
- To ensure some 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 security property.

Fragile Base Class Continued

The scenarios discussed so far involve overriding. Is it safer to just inherit and add new methods?

- Not without risk…
- Accidental method capture: A new release of the base class accidentally includes a method with the same name.

- All of the sudden
  - Your code does not compile because the new method in the base class has the same signature but a different return type.
  - Your new methods get involved in things you never thought about because the added method has the same signature and return type.

4.2.3.4 Taming Inheritance

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

- But, it is **not always the best tool** for the job.
- Used inappropriately, it leads to fragile software.

- The next few slides, discuss some **rules of thumb** for making **"good use" of inheritance**.

A Handful Dos and Don'ts

It is always safe to use inheritance within a package or another "whitebox" setting.
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** (see next slide).

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

Document for Inheritance

Design and document for inheritance
or else prohibit It.

Joshua Bloch, Effective Java

Classes Must Document Self-Use

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.

- Overridable method = non-final and either public or protected
- A class must document any circumstances under which it might invoke an overridable method.
  Invocations might come from background threads or static initializers.

Conventions for Documenting Self-Use

- 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 Self-Invocation Documentation

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.

The documentation makes explicit:

- Overriding `iterator()` will affect the behavior of `remove`.
- What the effect would be.

*) The example is copied from the specification for java.util.AbstractCollection

Conventions for Documenting Self-Use

- …
- The description starts with “This implementation …”.
- Indicates that the description **tells something about the internal working** of the method.

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

Conventions for Documenting Self-Use

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

- Yes and no!
- Keep in mind: There are two kinds of clients of a extensible class:
  - **Ordinary clients** create instances of the class and call methods in its interface (black-box use).
  - **Subclassing clients** extend the class via inheritance.

- Ordinary clients should not know such details.

  … At least as long as a mechanism for LSP is in place.
- Subclassing clients need them. That's their "interface"!

- Current documentation techniques and tools lack means of separating the two kinds of API documentations.

Provide and Document Hooks to Internals

- A class must document the supported hooks to its internals.
- These internals are irrelevant for ordinary users of the class.
- But, they are crucial for enabling subclasses to specialize the functionality in an effective way.

Example*) of Documenting Hooks

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.*

*) The example is copied from the specification for java.util.AbstractList

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.

- On the other hand, you must not provide too few.
- A missing protected method can render a class practically unusable for inheritance.

So, how does one decides about the protected members to expose?

How to Decide About Members to Expose?

- No silver bullet!
- Think hard, take your best guess, and TEST!
- **Test your class for extensibility before releasing them**.
  By writing subclasses before releasing it.
  At least one subclass should be written by someone other than the superclass author.

Constructors Must not Invoke Overridable Methods

Problem:

- An overridable method called by a constructor may get invoked on a non-initialized receiver.
- As a result a failure may occur.

Reason:

- The superclass constructor runs before the subclass constructor.
- The overridden method will get invoked before the subclass constructor has been invoked.
- The overridden method will not behave as expected if it depends on any initialization done by the subclass constructor.

4.2.4 Variations at the Level of Multiple Objects

Variations at the Level of Multiple Objects

- So far, we considered variations, whose scope is individual classes.

- In general, **the scope of a variation is a group of related classes**.
- Recall: **No class is an island**!

- Examples of such 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.

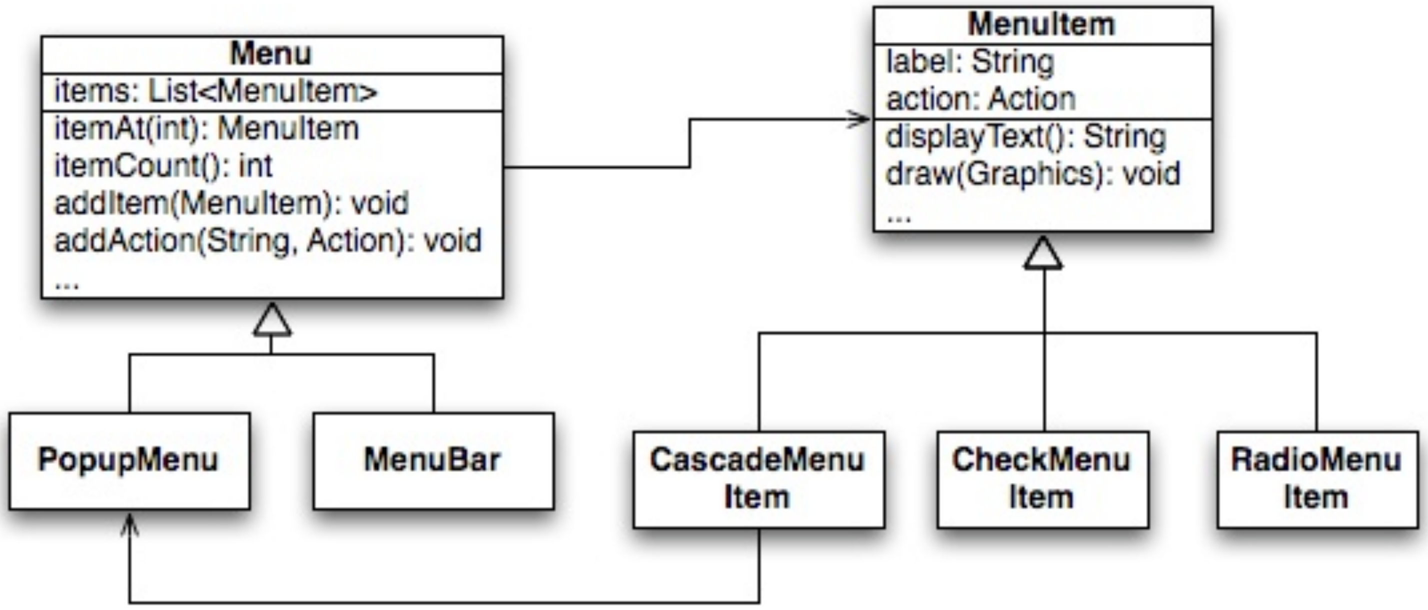
4.2.4.1 Illustrative Example: Window Menus

- 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.

Classes Involved in Implementing Window Menus

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

- A possible implementation of a menu structure is shown in the figure below and detailed in following slides.



Classes Involved in Implementing Window Menus

- 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` or 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.

Variants 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 of their fields and methods.

Variants 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 { ... }
```

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

Optional Features of Menus

There is a variety of optional features related to functionality of menus:

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

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 modularize variations of menu functionality.
- This technique has several problems.

  The problems are illustrated in the following by a particular example variation: Adding accelerator keys to menus.

Menu Items with Accelerator Keys

- 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 method `draw` 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
    - …

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

- The extension of menus with accelerator keys is implemented in class `MenuAccel`, a subclass of `Menu`:
  - adds the new method `processKey` for processing keys
  - overrides method `addAction` to ensure that the new item added for an action supports 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));
   }
   ...
}
```

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 ???
```

What about the subclasses of `MenuItem` for different variants of items?

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 ???
```

What about the subclasses of `Menu` for different variants of menus?

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

4.2.4.2 Combining Composite & Individual Variations

- In general, the problem is how to combine **two kinds of variations**.

- **Variations at the level of object composites**
- In our case: accelerator key support

- **Variations at the level of individual elements of the composite**
- In our case:
  - variations of menus, e.g., pop-up menus, menu bars, etc.
  - Variations of menu items, e.g., cascade menu items, check boxes, etc.

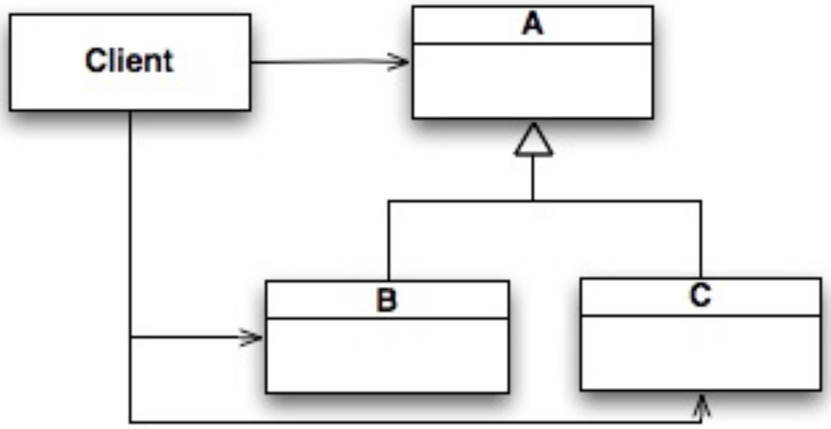
Combining Composite and Individual Variations

In languages with single inheritance such as Java, such a combination is non-trivial and leads to code duplication.

Multiple inheritance is no satisfactory solution either

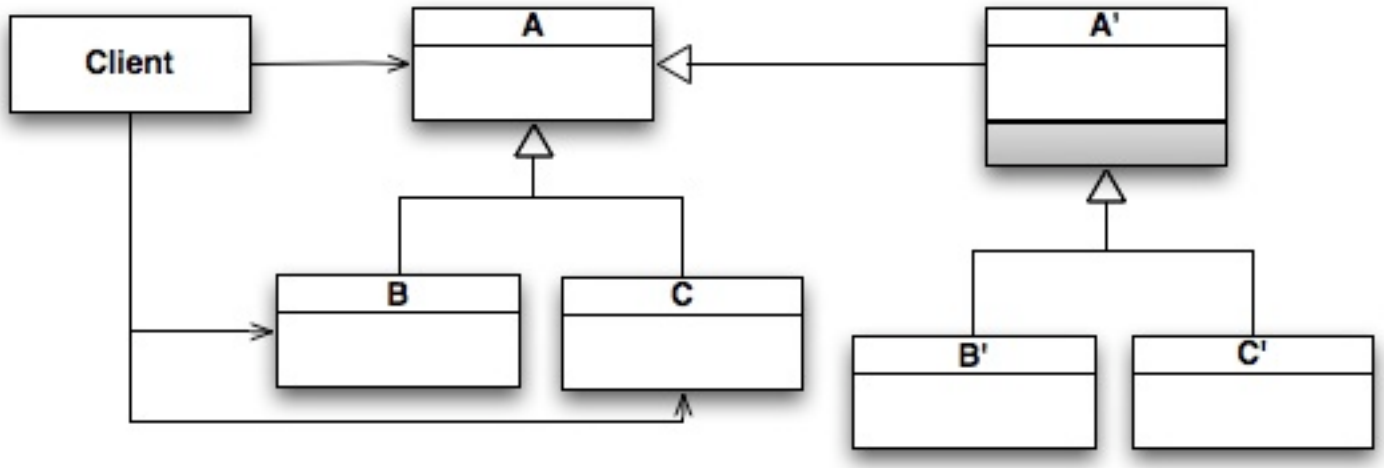
The Problem in a Nutshell

- There is a parent class `A` with subclasses `B` and `C` and there exist clients of `A, B, C`.
- We need to extend `A` (and 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.



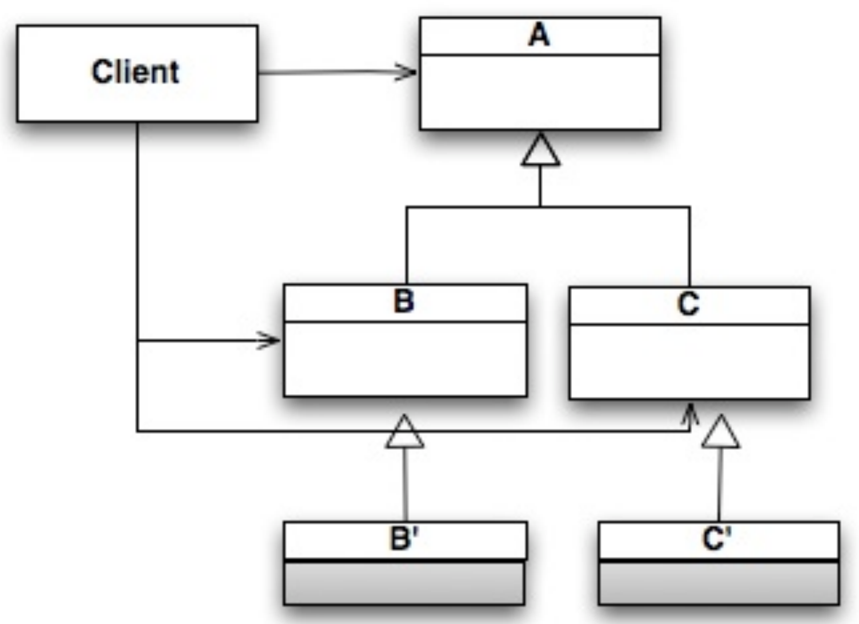
- There are two possibilities to add an optional feature to `A` incrementally without affecting clients in a single inheritance setting.
- In both cases code needs to be duplicated.

Alternative 1



- Implement the new feature in a subclass of `A`, called `A'`.
- Re-implement the "base" functionality of `B` and `C` in subclasses of `A'`, called `B'` and `C'`.
- Need to ensure that for every new variant of `A` defined in the future, say `D`, the corresponding `D'` is also defined.

Alternative 2



- Inherit from leaves (`B` and `C`) and define the new functionality twice.
- The implementation of the new feature is duplicated.
- Need to ensure that for every new variant of `A` elements defined in the future, say `D`, the corresponding `D’` is also defined.

Combining Composite and Individual Variations

Let us just assume that we do have some form of multiple inheritance and don’t bother about this issue further.

```
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 { ... }
```

Combining Composite and Individual Variations

- The design with **multiple inheritance** is not optimal too.

- It requires a set of 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 **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.

- Multiple inheritance has its own set of problems that make its usage troublesome
  - Semantics of diamond inheritance
  - How to resolve conflicts

4.2.4.3 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.
- **Covariant dependencies are emulated by type-casts**.

Non-Explicit Covariant Dependencies

```
abstract class MenuAccel extends Menu {
   boolean processKey(KeyStroke ks) {
     for (int i = 0; i < itemCount(); i++) {
        if (((MenuItemAccel) itemAt(i)).processKey(ks)) return true;
        }
   }
   vo
   }
   .
}

class
class
```

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`.

Non-Explicit Covariant Dependencies

```
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 ???
c
```

The design cannot guarantee that such a type cast will always be successful, because menu 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!

4.2.4.4 Instantiation-Related Reusability Problems

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

- For example, `MenuItem` is instantiated in `Menu.addAction(...)`.
- In `MenuAccel`, we must override `addAction(...)`, so that it instantiates `MenuItemAccel` rather than `MenuItem`.

```
abstract class Menu {
   ...
   void addAction(String label, Action action) {
              items.add(new MenuItem(label, action));
   }
```

```
abstract class MenuAccel extends Menu {
   ...
   void addAction(String label, Action action) {
        items.add(new MenuItemAccel(label, action));
   }
   ...
}
```

Instantiation-Related Reusability Problems

- Instantiation code can be spread all over the application.
- 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)
      ...
   }
...
}
```

A contributor for operations on files

Instantiation-Related Reusability Problems

- In some situations, overriding of instantiation code can cause 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`.

Instantiation-Related Reusability Problems

Do you know a pattern that might help with the instantiation-related problems?

Factories for Instantiating Objects

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

Factories for Instantiating Objects

```
class FileMenuContrib implements MenuContributor {
   void contribute(Menu menu, MenuFactory factory) {
      MenuItem openWith =
               factory.createCascadeMenuItem(”Open With”);
      menu.addItem(openWith);
      MenuItem openWithTE = factory.createMenuItem(...);
      openWith.addItem(openWithTE);
      ...
      MenuItem readOnly = factory.createCheckMenuItem(...);
      menu.addItem(readOnly)
      ...
   }
 ...
}
```

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

The code of `FileMenuContrib` can be reused with different variations of menu functionality, by using it with different factory implementations.

Deficiencies of the Factory Pattern

- The **infrastructure** for the design pattern **must be** manually **implemented and maintained**.

- **Correct usage** of the pattern **cannot be** completely **enforced**:
  - there is no guarantee that classes are instantiated exclusively over the factory methods,
  - There is no guarantee that only objects instantiated by the same factory are used together.

Deficiencies of the Factory Pattern

- **No good solution exists for managing the reference to the abstract factory**.

- The factory can be implemented as a Singleton for convenient access to it within entire application.
- But 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.

4.2.4.5 Recapitulating

- General agreement in the early days of OO:
  Classes are the primary unit of organization.
- **Over years, it turned out that sets of collaborating classes are better units of organization.**

- An extension will generally affect a set of related classes.
- 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.
- But, this solution has several drawbacks …

Inheritance does not appropriately support OCP with respect to changes that affect a set of related classes!

Language Mechanisms for Sets of Related Classes

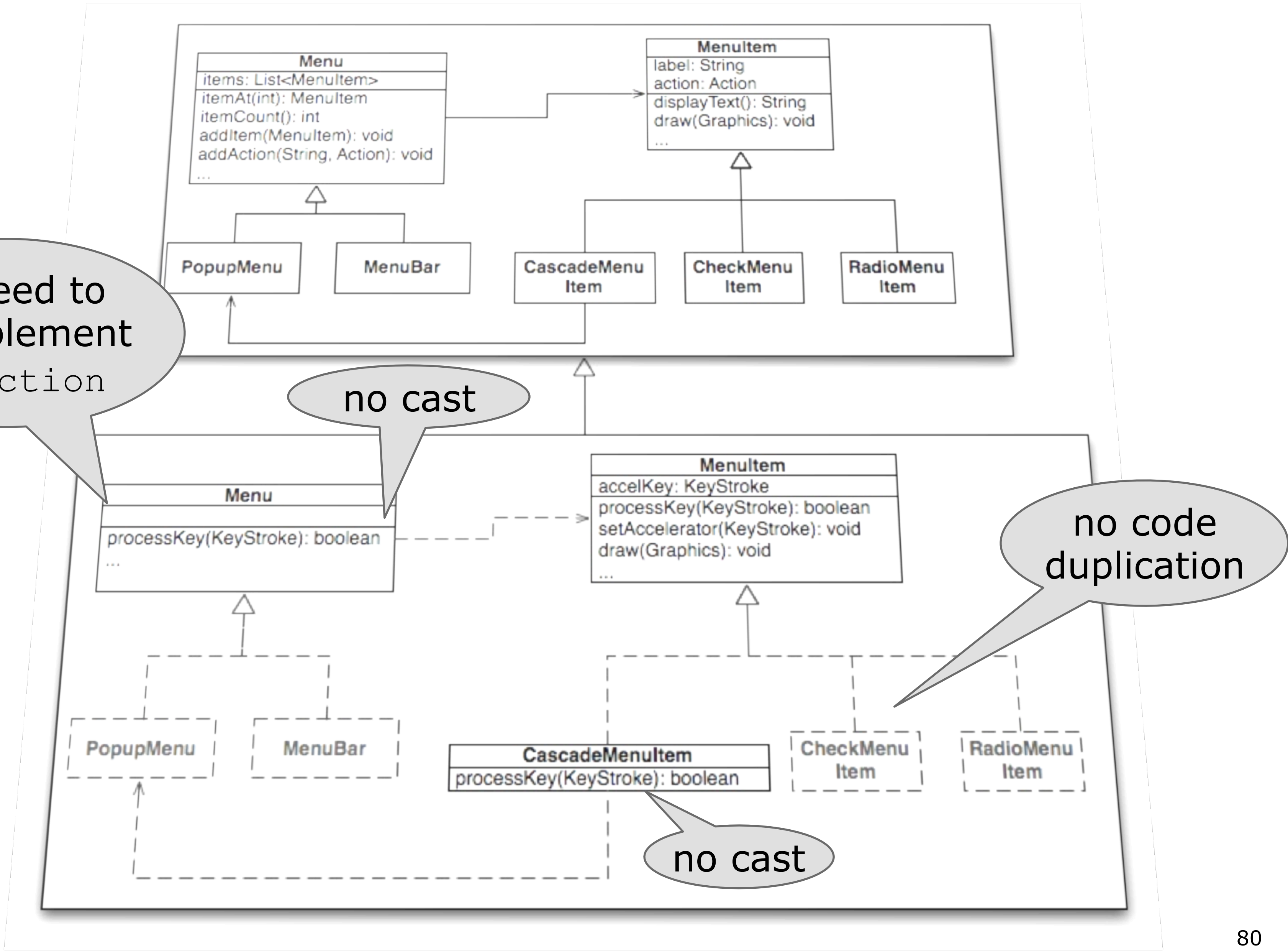
- Mainstream OO languages have only insufficient means for organizing collaborating classes: packages, name spaces, etc.
- These structures have serious problems:
  - No means to express variants of a collaboration.
  - No polymorphism.
  - No runtime semantics.

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

What we Actually Want

- Ideally, we would like to model the accelerator key functionality by the difference to the default menu functionality.
- Be able to define behavior that is polymorphic with respect to the particular object group variation.
- **Incremental programming at the level of sets of related classes.**
  In analogy to incremental programming at the level of individual classes enabled by inheritance.

What we Actually Want



déjà vu?

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

Base configuration

```
abstract class Location {
}
abs                  ion extends Location {

}
cla
}
cla



}
cla



}
...
House house
House house
...
```

Shutter control

```
abstract class Location {
    a                  > shutters();
}

abst
    I
}

clas
    I
    I
}
```

Lighting control

```
abstract class Location {
    abstract List<Light> lights();
}

abstract class CompositeLocation {
    List<Light> lights() { ... }
}

class Room {
    List<Light> lights;
    List<Light> lights() { return lights; }
}
```

4.2.4.6 Case Study: Java AWT and Swing

- **Question**: Does the described problem show up in large-scale software designs and frameworks?

- **Answer**: Yes it does.
- To show that the problems are real, let us take a look at the case of Java AWT and Swing*).

*) Slides in this subsection discuss material from the paper: *Bergel et al, Controlling the Scope of Change in Java, International Conference on Object-Oriented Programming Systems Languages and Applications 2005*

About the Development of Swing

- AWT is a GUI framework that was included in the first Java release.
  It is close to the underlying operating system, therefore only a small number of widgets are supported to make code easier to port.

- The Swing framework is built on top of AWT:

- Swing uses subclassing to extend AWT core classes with Swing functionalities:
  - A "pluggable look and feel" and double buffering.
    Swing-specific self-description functionality:
    Attribute `accessibleContext` in Swing widgets refers to a description of the component.
    Swing-specific support for double buffering to provide smooth flicker-free animation (methods `update()`, `setLayout()`, etc.).

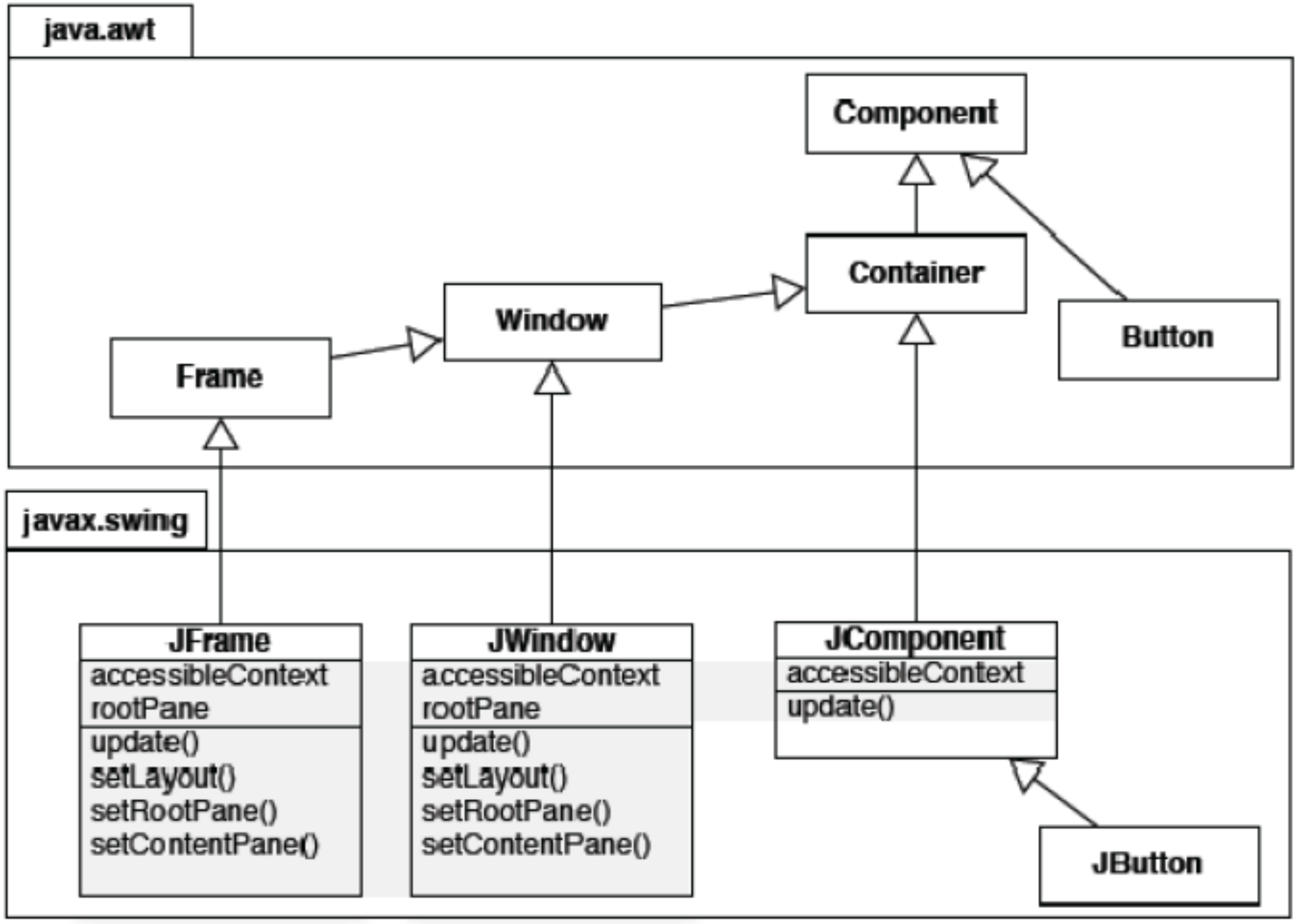
- Swing adds more widgets.

About the Development of Swing

“Consider, for example, the development of Swing, a GUI package for Java that was built on top of the older AWT package.
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.

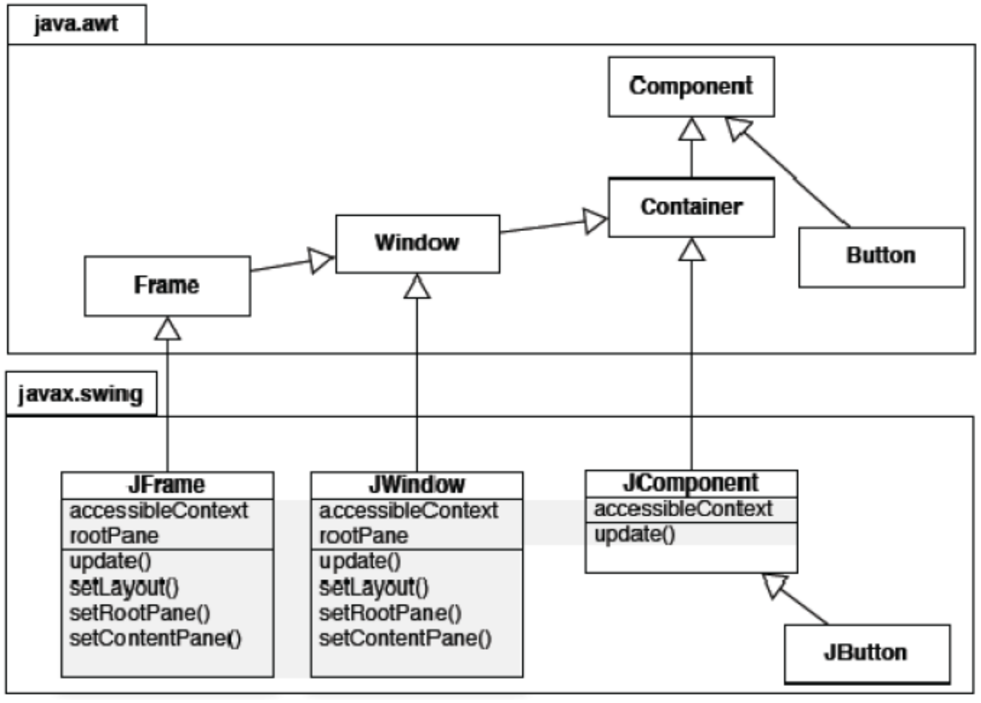
Such a refactoring, however, was not an option, and we can witness various ***anomalies in Swing****, such as* ***duplicated code, sub-optimal inheritance relationships, and excessive use of run-time type discrimination and downcasts.****”*

The Design of AWT and Swing



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

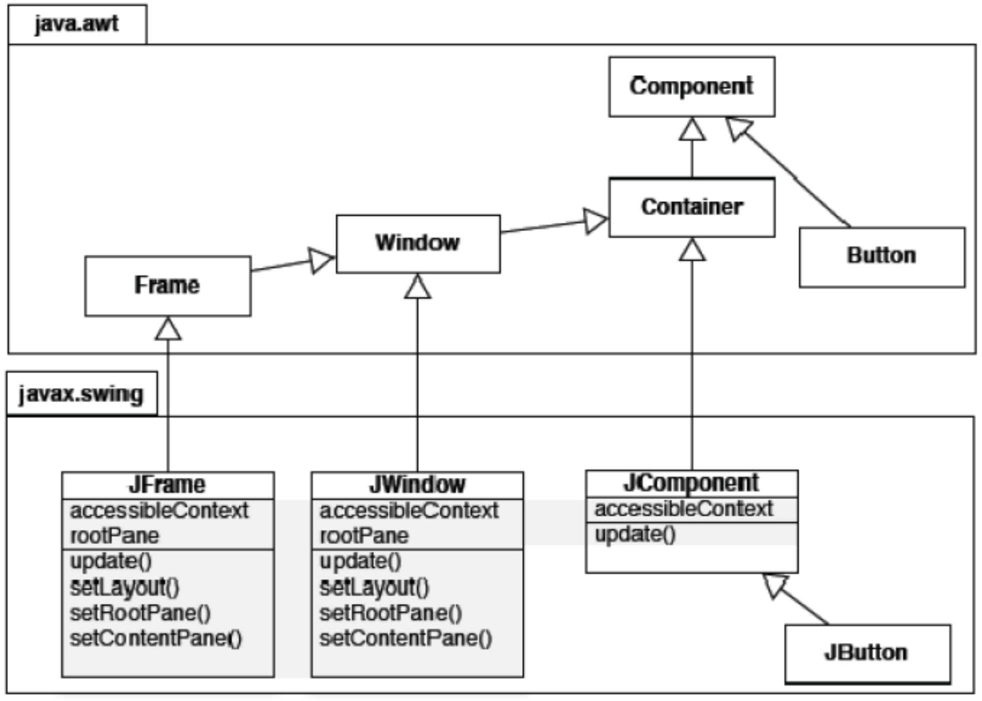
Code Duplication



- **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 LOC duplicated!
- **43% of `JWindow` reappears as 29% of `JFrame`.**

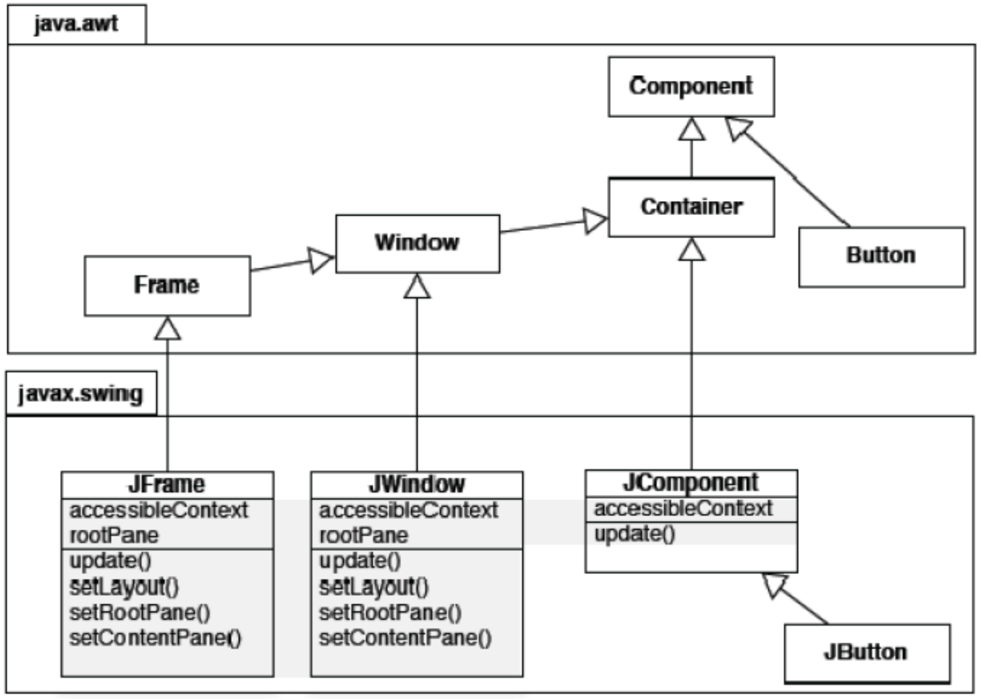
Breaking Subtyping Inheritance



- 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`.

Explicit Type Checks and Casts



- A Swing `Component` is a `Container` for other `Component`s.
  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: No Type Checks and 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 getComponents(int index) {
        return component[index];
    }

}
```

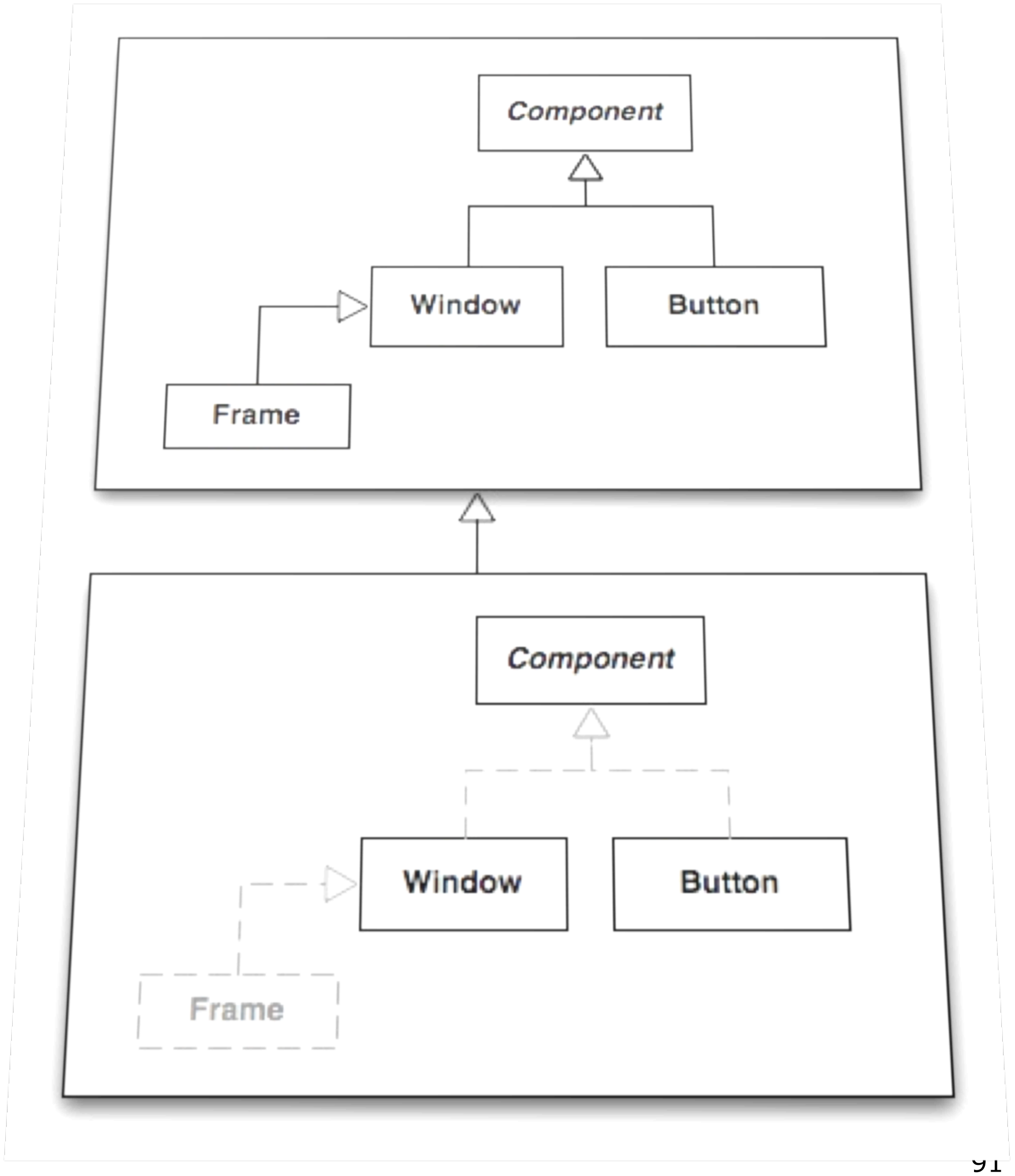
Swing Code: Type Checks and Casts

```
public class JComponent extends Container {

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

}
```

What We Really Want

Similar to defining menus with accelerator keys by the delta to “normal” menus,

again we wish to be able to extend the set of collaborating classes as if it was a single class!



4.2.4.7 Case Study: Game Development

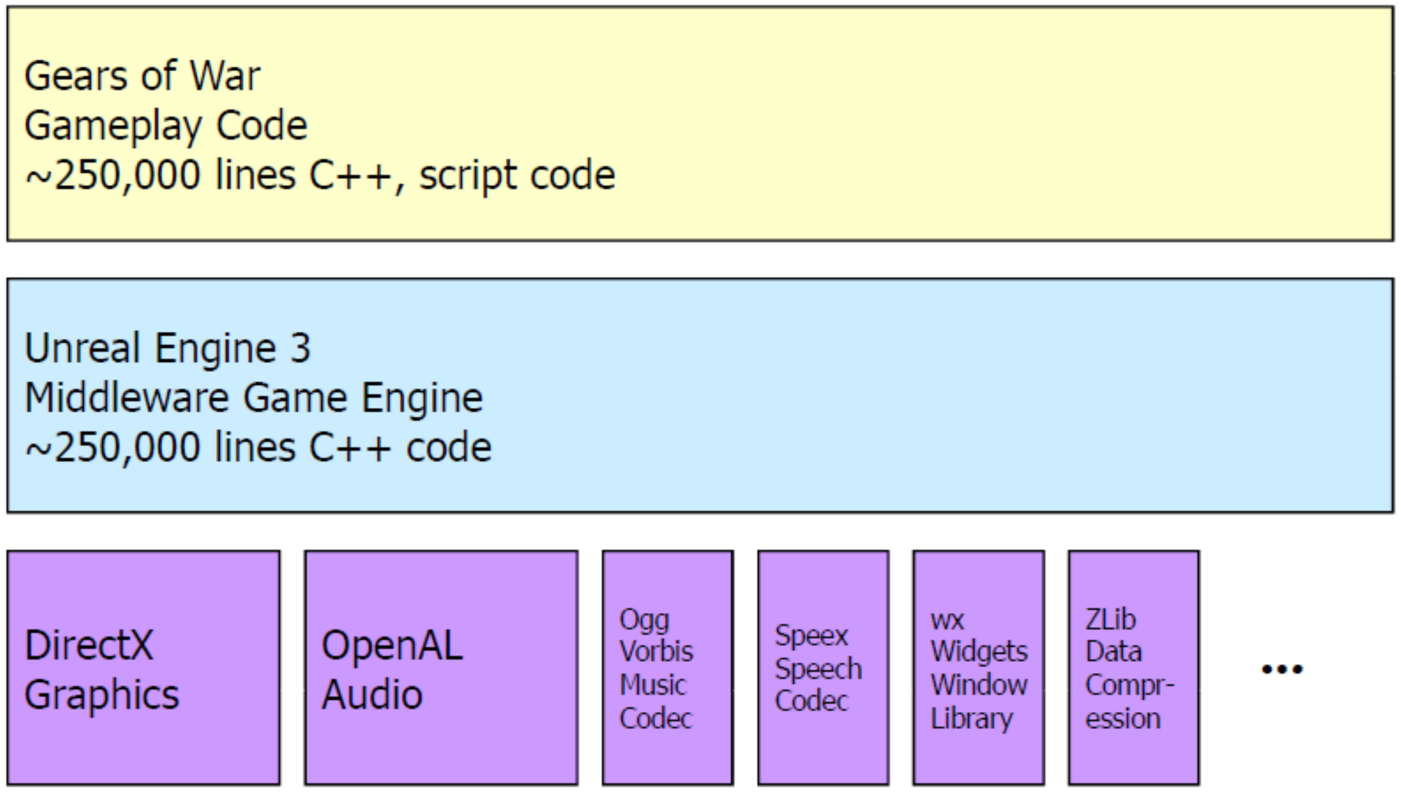
The slides in this subsection are “borrowed” from a keynote by Tim Sweeney, Founder of Epic Games,
at the ACM Conference on Principles of Programming Languages ’06.

The Next Mainstream Programming Language:

A Game Developer's Perspective

Tim Sweeney
Epic Games

Software Dependencies



What are the hard problems?

- Performance
  - When updating 10,000 objects at 60 FPS, everything is performance-sensitive
- Modularity
  - Very important with ~10-20 middleware libraries per game
- Reliability
  - Error-prone language / type system leads to wasted effort finding trivial bugs
  - Significantly impacts productivity
- Concurrency
  - Hardware supports 6-8 threads
  - C++ is ill-equipped for concurrency

Unreal's game framework

Gameplay module

Base class of gameplay objects

Members

```
package UnrealEngine;

class Actor
{
      int Health;
      void TakeDamage(int Amount)
      {
            Health = Health – Amount;
            if (Health<0)
                  Die();
      }
}

class Player extends Actor
{
      string PlayerName;
      socket NetworkConnection;
}
```

Game class hierarchy

Generic Game Framework

- Actor
  - Player
  - Enemy
  - InventoryItem
    - Weapon

Game-Specific Framework Extension

- Actor
- Player
  - Enemy
    - Dragon
    - Troll
  - InventoryItem
    - Weapon
      - Sword
      - Crossbow

Software Frameworks

- The Problem:
  Users of a framework
  want to extend the functionality
  of the framework's base classes!

- The workarounds:
  - Modify the source
    ...and modify it again with each new version
  - Add references to payload classes, and dynamically cast them at runtime to the appropriate types.
  - These are all error-prone:
    Can the compiler help us here?

What we would like to write...

Base Framework

```
package Engine;

class Actor
{
        int Health;
        …
}
class Player extends Actor
{
        …
}
class Inventory extends Actor
{
        …
}
```

Extended Framework

```
Package GearsOfWar extends Engine;

class Actor extends Engine.Actor
{
        // Here we can add new members
        // to the base class.
        …
}
class Player extends Engine.Player
{
        // Thus virtually inherits from
        // GearsOfWar.Actor
        …
}
class Gun extends GearsOfWar.Inventory
{
        …
}
```

The basic goal:

To extend an entire software framework's class hierarchy in parallel, in an open-world system.

4.3 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**.

So…

- Is inheritance a good idea after all?
  - What would life without inheritance be like?
    - E.g., how do functional programmers deal with variability?
  - Could improved variants of inheritance help?
    - Mixins, traits, prototypes, virtual classes, mixin layers, delegation layers, abstract types, …

- What *is* inheritance really?
  - A mechanism to create subtypes?
  - A mechanism to reuse-and-patch code?
  - A mechanism to model specialization/generalization in the real world?
  - …