Stepwise Feature Introduction

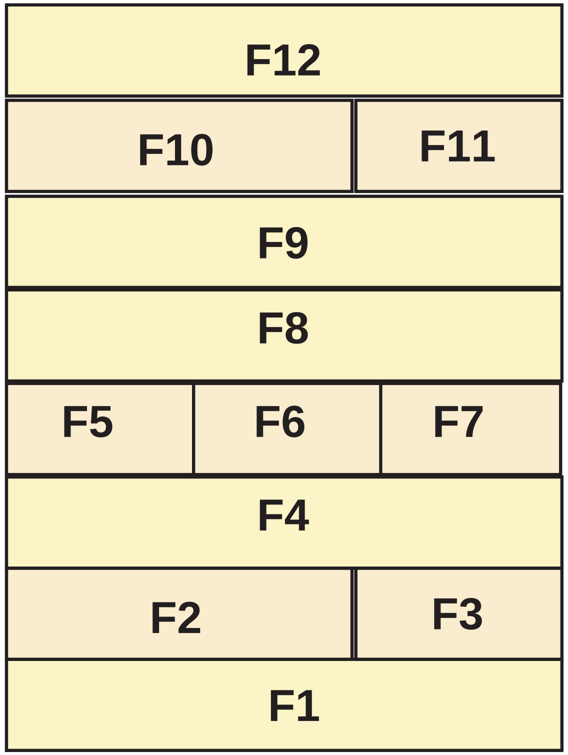
R. J. R. Back
Turku Centre for Computer Science (TUCS) and
Åbo Akademi University

Stepwise feature introduction

We consider here a specific method for software construction, where

- the software is constructed in **thin layers**,
- adding one **feature** at a time to the software,
- checking that previously added features are **preserved** when adding a new feature, and
- the layer structure is **refactored** whenever needed.

Refer to this as **stepwise feature introduction**.

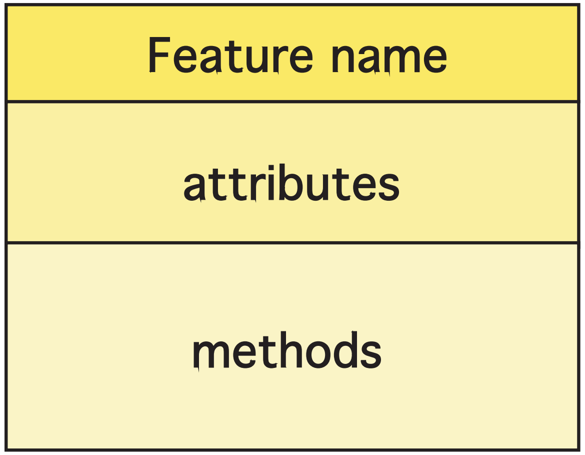


Features

A software component provides a *service*.

- The service consists of a collection of *(service) features*.
- A feature will be implemented as a class.
- This class (usually) extends one or more other classes
- A class introduces a new feature while at the same time inheriting all the features introduced in the classes that it extends.

We use **(multiple) inheritance** as the extension mechanism for features.



Example: A text widget

Simple text widget component: display the text, react to mouse clicks, insert new text, delete text, select text, move the insertion point (by clicking the mouse or using the cursor keys.), etc.

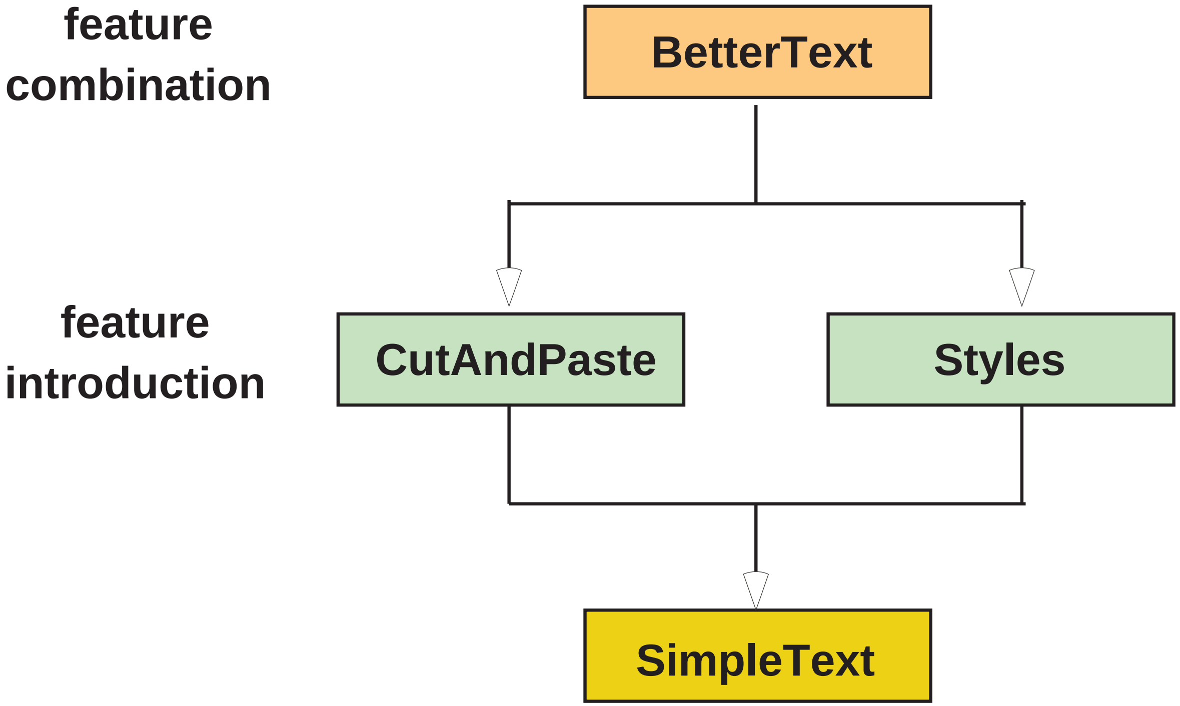
Additional features:

saving Save the text in the widget to a file, or replace the text in the widget with the contents of a file.

cut and paste Cut (or copy) a selected piece of text into the clipboard, and paste the contents of the clipboard into the place where the insertion cursor is.

styles Format pieces of text in the widget, e.g. by changing the selected text to boldface, italics, underlined, or colored text.

Extending the text widget



- Add cut and paste to text widget, without compromising earlier features
- Add styles to text widget, without compromising earlier features
- Combine the two extensions in a better text widget.

Managing feature interaction

Simply defining BetterText as an extension of CutAndPaste and of Styles (by multiple inheritance) will not do:

- CutAndPaste will only copy and paste ascii text
- Formatting is then forgotten when copying and pasting in BetterText.

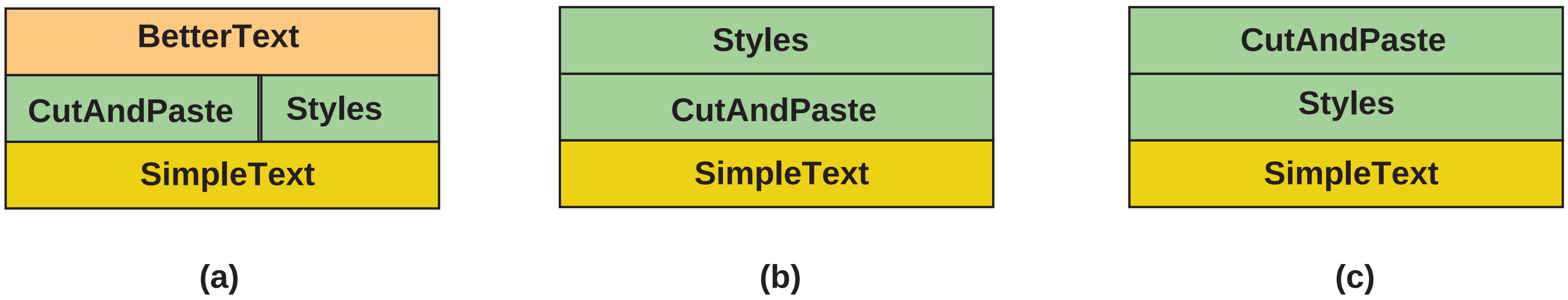
BetterText has to redefine cut, copy and paste operations so that formatting information is also preserved.

Separation of concerns

Layering allows us to separate our concerns:

- The features are added one by one
- Features can be introduced in parallel, working out the mechanism required by each feature separately first (**feature introduction**)
- The problems arising from the feature interactions can be addressed later, as a separate step (**feature combination**)
- Feature combination is easier to do when the mechanisms for the features to be combined have been worked out and are open for inspection.

Alternative layering



BetterText

CutAndPaste Styles

SimpleText

(a)

Styles

CutAndPaste

SimpleText

(b)

CutAndPaste

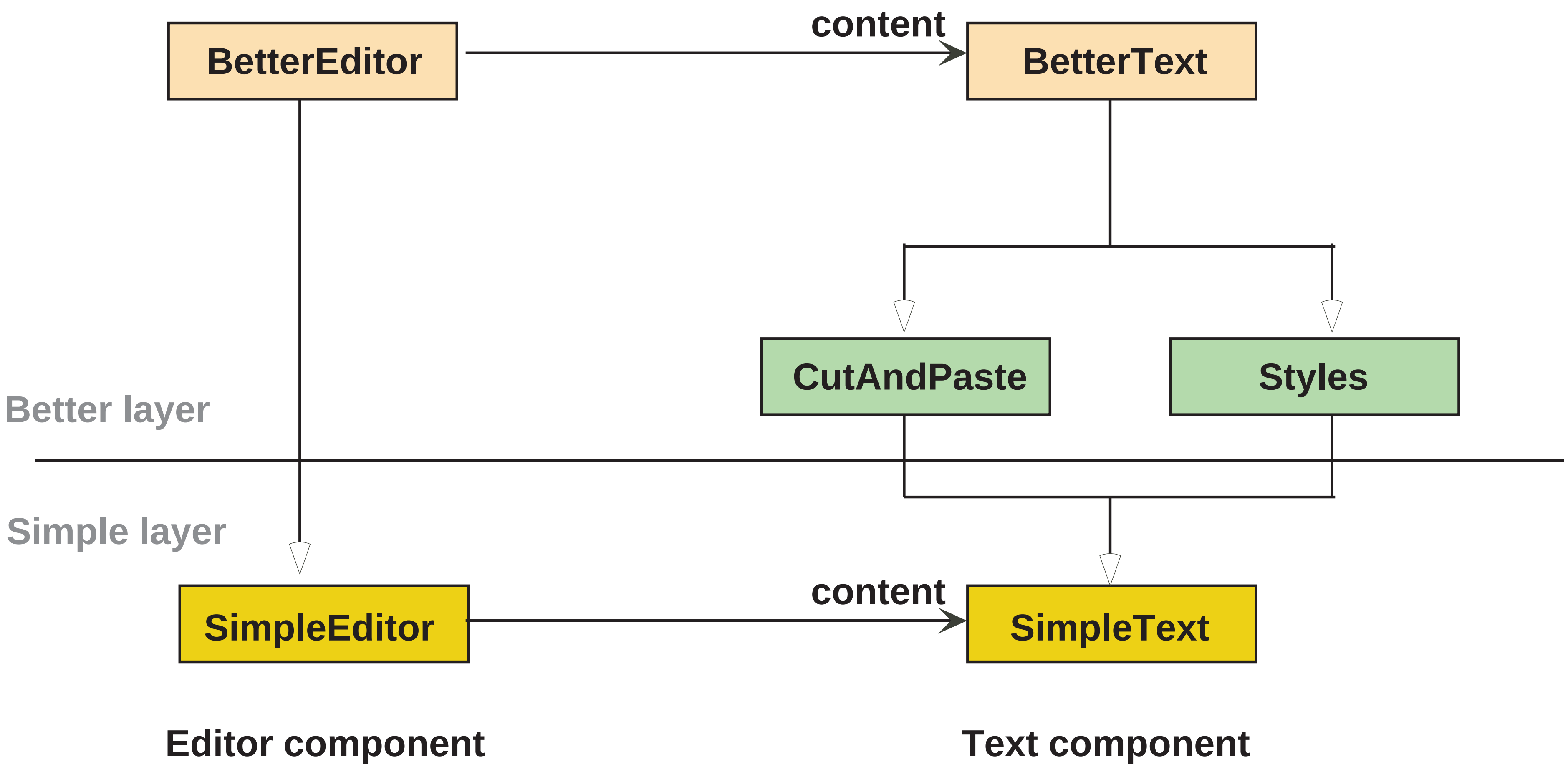
Styles

SimpleText

(c)

- (a) avoids an arbitrary choice between which feature to introduce first, and allows us to consider the two features in separation.
- (b) and (c) introduce one feature before the other, avoiding the combination step.
- The resulting hierarchy is simpler in (a), but this can happen at the expense of making the second feature more difficult to introduce.

Layers



- The text widget is extended together with an editor, that displays the text widget and has menus for manipulating it.
- There are two layers in this design: the *Simple* layer and the *Better* layer

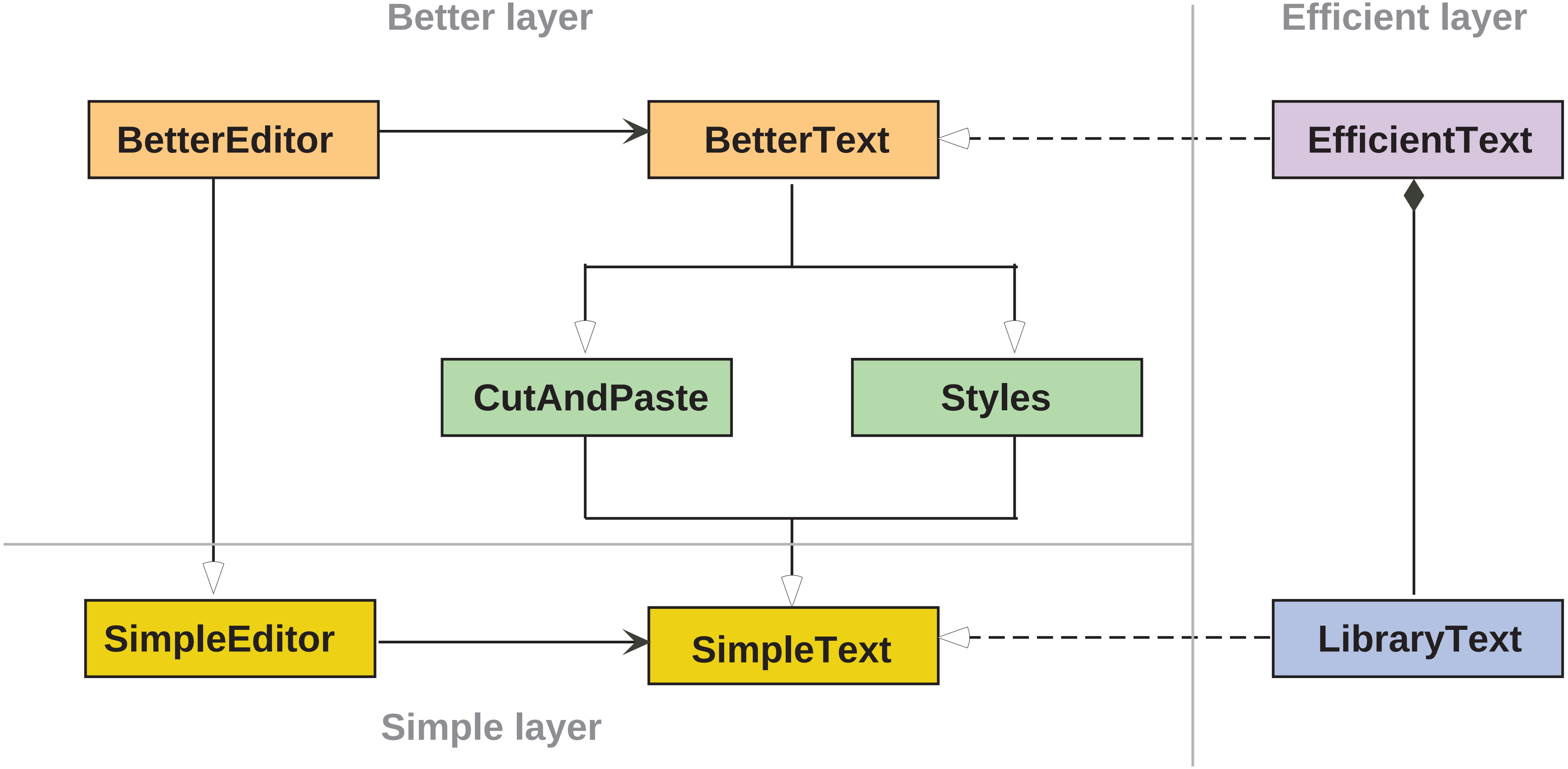
Layering property

This structure satisfies the layering princple :

- The SimpleEditor together with SimpleText forms a layer that can function without the upper layer
- The BetterEditor together with BetterText, CutAndPaste and Styles forms an upper layer, that gives the full functionality of the editor
- The SimpleEditor can call BetterText, but it cannot make use of the new features (cut-and-paste or styles), because it does not know about these.
- We need to build BetterEditor in order to use the new features of BetterText.
- BetterEditor cannot use SimpleText, because the functionality it assumes of the text widget is not implemented in SimpleText.

BetterText as specifications

We may consider BetterText as a **specification** of a more efficient class (maybe from some class library).



This gives us a **layered** specification of a software component.

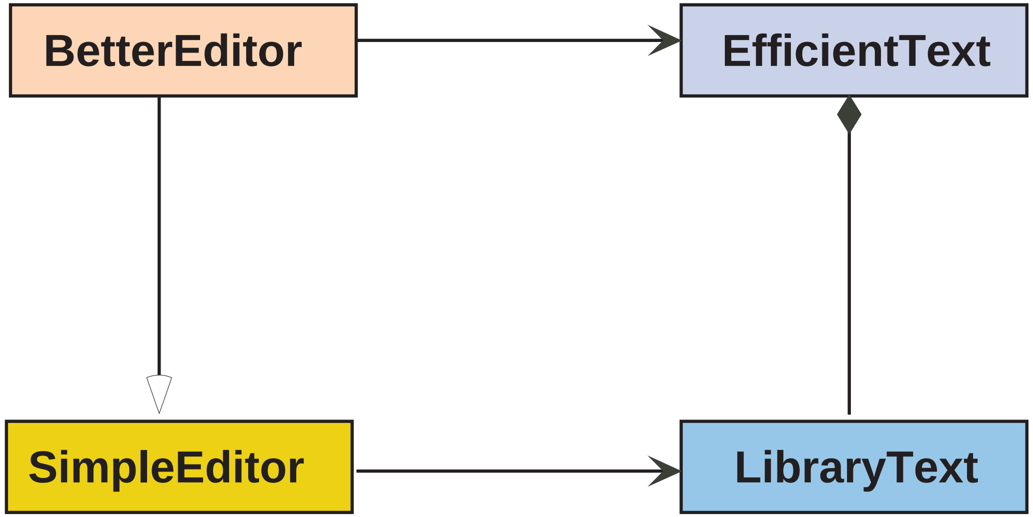
Layered specifications

This gives us a **layered** specification of a software component.

- BetterText is implemented by EfficientText
- EfficientText is a wrapper for LibraryText
- LibraryText is taken directly from a class library
- SimpleText can be implemented directly by LibraryText

Using monotonicity

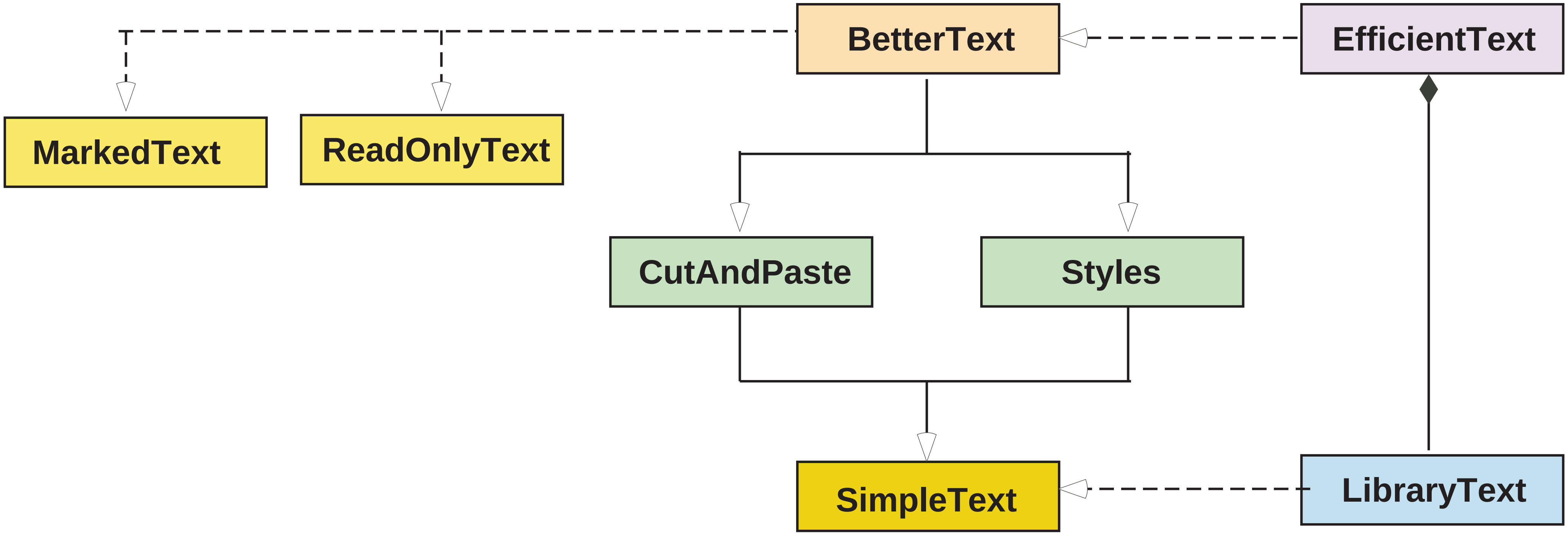
Simplified diagram (BetterEditor uses EfficientText directly):



BetterText as implementation

BetterText is an implementation that supports different specifications or **views**:

- ReadOnlyText – cannot change text
- MarkedText – can highlight (change styles) but not change text



Extreme programming

Extreme programming is a collection of techniques and practices that together form a software process for small team system development (Beck and others). It emphasizes

- short iteration cycles in the development,
- a concentration on producing code rather than documentation,
- avoiding to plan for future extensions,
- frequent testing and integration of software during development,
- frequent refactoring of software structure,
- pair-programming, and
- on-site customers,

among other things.

Stepwise feature introduction process

Extreme programming also emphasizes the incremental construction of software.

The **planning game**:

- the customers list and rank the features they need in the software,
- the programming team estimates the time and other resources needed to implement each feature
- Each iteration cycle is carried out in a fixed time
- Iteration includes only those features that the customer has deemed to be most important and which all can be implemented in the allocated time

Stepwise feature introduction can be seen as a complement to the extreme programming process:

- it describes more precisely how to structure the software and build the components
- the components are developed in the incremental fashion that extreme programming proposes.

Criticism of extreme programming

Extreme programming has also been seen as a new coming of hackerism:

- it de-emphasizes documentation and careful design of the software, and
- it emphasizes the coding part of the software process.

This is not a necessary feature of extreme programming, it is possible to add

architecture, documentation and correctness concerns

as central parts of the incremental software construction process.

Requirements

Previous correctness criteria guarantee that the system is consistent, but it may still be doing something completely different from what we intended. Must also take system requirements into account.

Requirements are modelled by a **customer** class. The customer checks that the service it gets is what it expects.

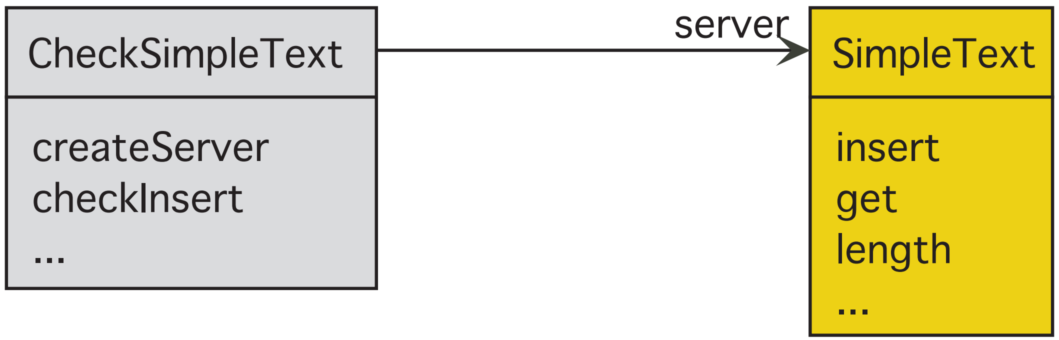
Express **service requirement** R by

$$R \quad = \quad [P]\,;S\,;\{Q\}$$

- P and Q are conditions (predicates) on the program variables (object attributes).
- The **assume statement** $[P]$ means **if** P **then** skip **else** succeed **fi**.
- S is a statement that describes the requested behavior.
- The **assertion statement** $\{Q\}$ means **if** Q **then** skip **else** fail **fi**.

Example customer

An customer class CheckSimpleText for the SimpleText class could be structured as follows:



Method checkInsert expresses a requirement on inserting text in SimpleText.

```
def checkInsert(self, t,s,i):
    $[0 \leq i \leq |t|]$
    server:= self.createServer(t)
    server.insert(i,s)
    s1:=server.get()
    $\{s1 = t[: i] + s + t[i :]\}$
```

```
def createServer(t):
    return SimpleText(t)
```

Requirements and unit tests

Requirements are similar to *tests*, except that they cannot be executed automatically if they have parameters.

A requirement that does not require any parameters can be seen as a *test*.

Example test method:

```
def testInsert(self):
    self.checkInsert("abcdefg", "XYZ",2)
```

Requirements for CutAndPaste

Define customer class CheckCutAndPaste, with e.g. following requirement as a method:

```
def checkCutFollowedByPaste(self,t,i,j,k):
    [$0 \le i \le j \le |t| \wedge 0 \le k \le |t| - j + i$ ]
    server:= self.createServer(t)
    server.cut(i,j)
    server.paste(k)
    s1:= t[:i]+t[j:]
    s2:= s1[:k]+t[i:j]+s1[k:]
    {server.get() = s2}
```

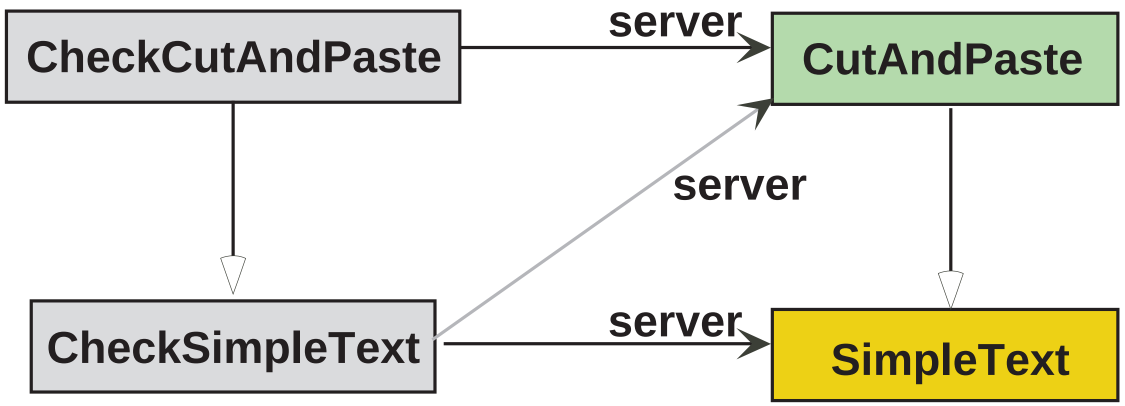
```
def createServer(t):
    return CutAndPaste(t)
```

Checking that old requirements still hold

All requirements for SimpleText should also hold for CutAndPaste

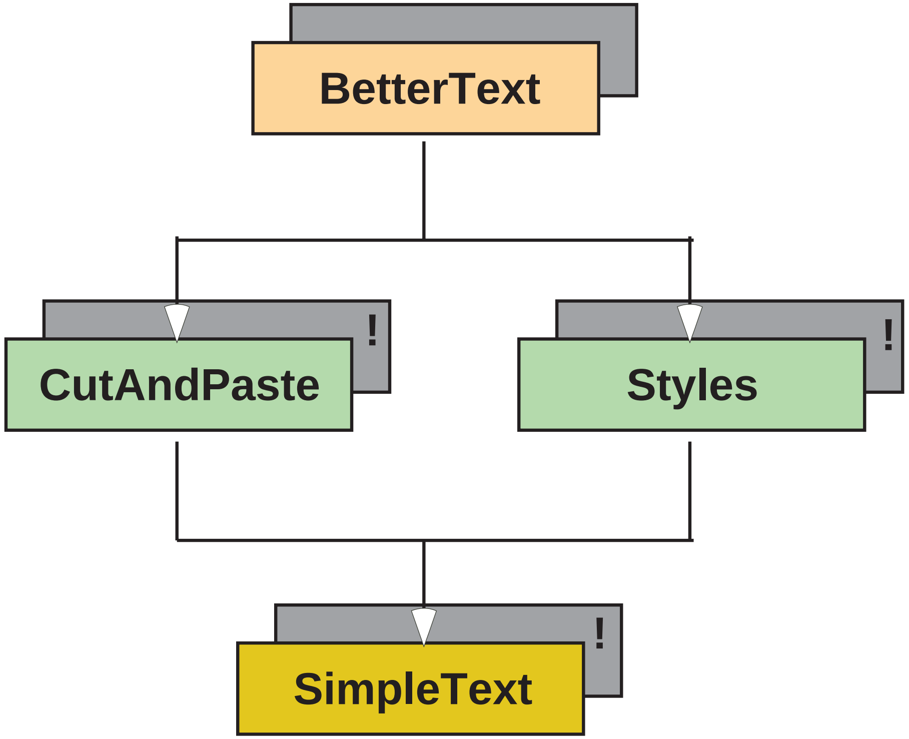
We therefore define the class CheckCutAndPaste to be a subclass of CheckSimpleText, so that it inherits all the requirements expressed as methods in CheckSimpleText.

Old methods will be applied to an instance of class CutAndPaste, rather than to an instance of class SimpleText, because we have redefined createServer in CheckCutAndPaste.



Requirements structure

- The customer classes is structured by inheritance in the same way as the server classes.
- No need to indicate inheritance structure explicitly, can use shadows to indicate customer classes.
- Exclamation mark on customer class indicates that the associated requirements are satisfied.



Steps in software construction

The stepwise feature introduction method defines the layering structure of the software, but not the order in which the different components are built (except the obvious one, that they should be built from bottom up).

In particular, we have not said when requirements and tests should be written and when they should be checked.

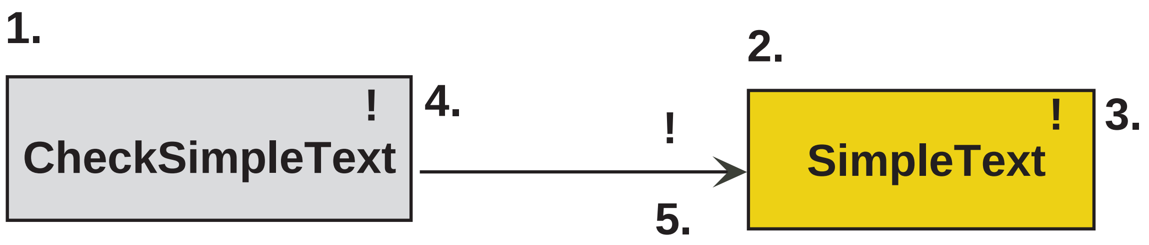
From the correctness point of view, it does not matter in which order the customer classes and the service classes are written, as long as consistency can be proved for all classes and correctness for all arrows.

Extreme programming prescribes that a tester (unit test class) for a class should be written before the class itself is written. Also, all tests should be executed automatically after each iteration cycle in the software development cycle.

We can generalizes the testing approach in extreme programming: requirements should be written before the code.

Constructing the text editor: base layer

1. Construct the class CheckSimpleText. It expresses the requirements for a simple text widget..
2. Construct the class SimpleText, using the requirements and tests in CheckSimpleText as design guidelines.
3. Check that SimpleText is internally consistent.
4. Check that CheckSimpleText is internally consistent and
5. Check that CheckSimpleText respect SimpleText .



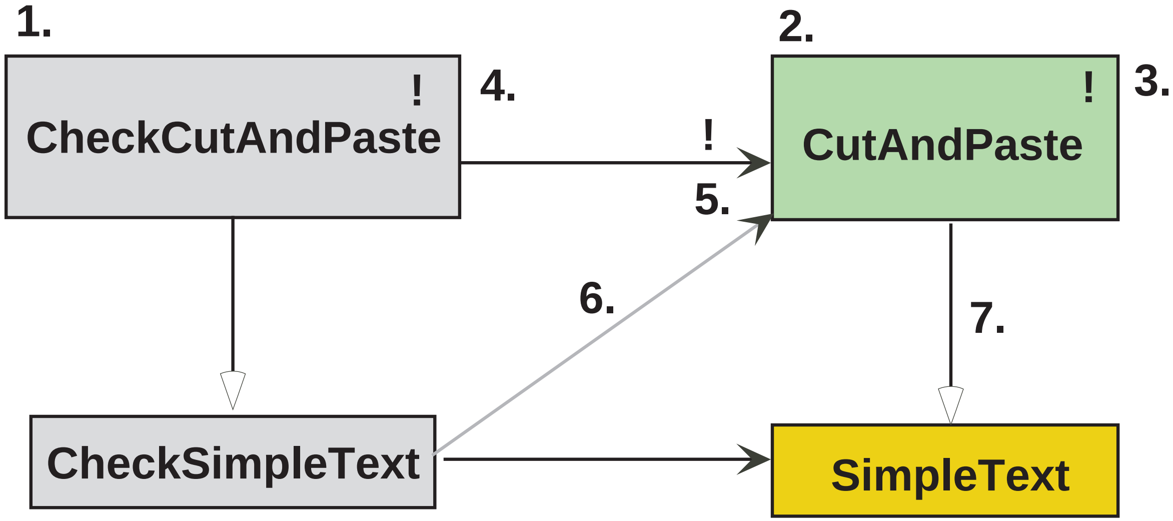
Constructing the text editor: next layer

Apply steps 1 - 5 to the classes CheckCutAndPaste and CutAndPaste. In addition:

6. Check that the requirements for SimpleText are still satisfied, after the cut-and-paste feature has been introduced.

7. Check that CutAndPaste is a superposition refinement of SimpleText.

Step 7 implies step 6, but requires a proof, whereas step 6. can be partly done by testing.



Design by contract

We assume that the following basic information is given for each class

- a **class invariant** that describes the legal states of an instance of the class
- a **precondition** for each method, describing the states in which the method may be called, and
- a **postcondition** for each method, describing properties of the state that holds after the method has been executed

The postcondition is not strictly neccessary, but it may be useful in order to analyze the effect of a method call.

Correctness requirements

We need to check that each class

- is **internally consistent**,
- **respects** the other classes that it uses,
- **preserves the features** of the classes it extends, and
- **satisfies its requirements** .

Internal consistency

The initialization of the class must establish the class invariant.

In addition, each method has to

- preserve the class invariant,
- preserve its loop invariants,
- establish its postcondition, and
- terminate,

whenever the precondition of the method is satisfied initially.

Respect

The class needs to satisfy

- the precondition of each method called from the class, and
- the precondition of each operation performed on values of basic types.

Preserving features

A method in a class that overrides a method of an extended class must behave essentially as the original method.

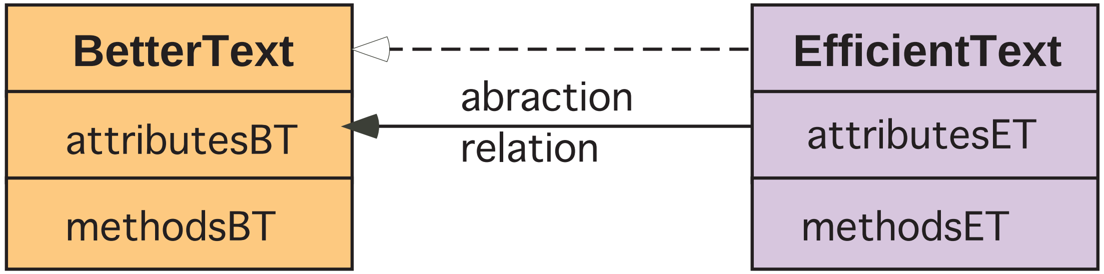
A customer using the extended class may not notice that it is using the extending class method.

This amounts to requiring that the extending class is a **superposition refinement** of the original class.

An arbitrary extension of a class is not automatically a superposition refinement, it has to be designed and proved to be one.

Correctness concerns

- Need to show that the implementation is a **data refinement** of the specification.
- Requires an abstraction function or relation from the implementation to the specification.
- A data refinement can change the collection of attributes that are used to represent state in a class, a superposition refinement can only add ne attributes.
- Usage is monotonic with respect to data refinement: can replace a class by its data refinement without changing the observed behavior of the system.



Satisfying requirements

The feature has been introduced for a specific reason, i.e., we have certain requirements that need to be realized by the feature.

We therefore also need to show that the feature does in fact satisfy the requirements given for it.

A feature extension may satisfy all the previous correctness criteria, but still not be what we want.

Correctness of software

In carrying out these proof steps, we are allowed to assume that all features on lower layers do satisfy their corresponding requirements.

If we prove that each feature extension satisfies these correctness conditions, then by induction, it will follow that the software system as a whole will satisfy these four correctness conditions.

Note that the final software system will be represented by one feature extension class, for which we then have shown the above properties.