



THE UNIVERSITY
OF AUCKLAND

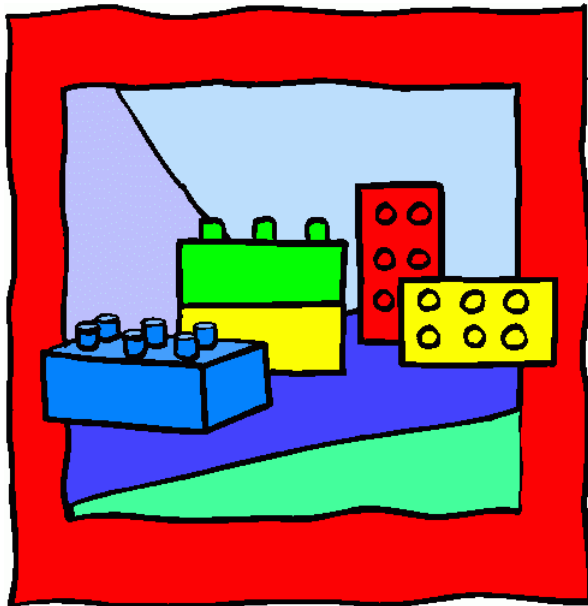
COMPSCI 230

Software Design and Construction

Design

2013-05-01

Design Principles



*LEGO is not a toy.
It's a way of life.
(Mike Smith)*

Separation of Concerns

How to deal with complexity in a system?

Separation of concerns (SoC)

- Separate issues (**break down** large problems into pieces) and concentrate on **one at a time**
- Break a program into **distinct features** that overlap in functionality as little as possible
- **Concern**: a **piece of a program**, usually a feature or a particular program behavior



Examples

- Separate concerns into **classes and methods**
- Separate **data from UI**, and UI from application logic
- Service-Oriented Architecture (SOA):
split up functionality into different **(web-) services**

Modularity

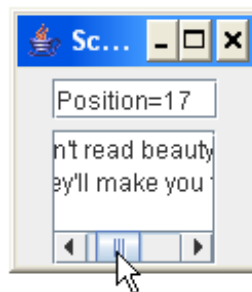
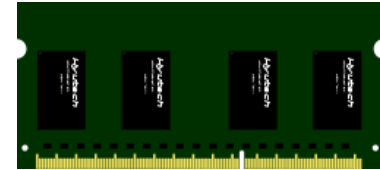
Complex systems can usually be **divided into simpler pieces** called modules

Module: self-contained component of a system

- Has a **well-defined interface** to other modules
- **Separates its interface** from its implementation

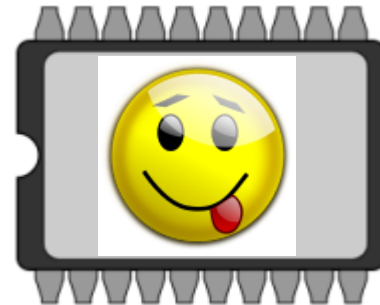
Modularity can be used on different levels:

- **Classes** that implement a well-defined **interface**
- **Packages** with classes and methods (and other types)
- Whole **programs** (e.g. command-line "pipes & filters")



Advantages of Modular Systems

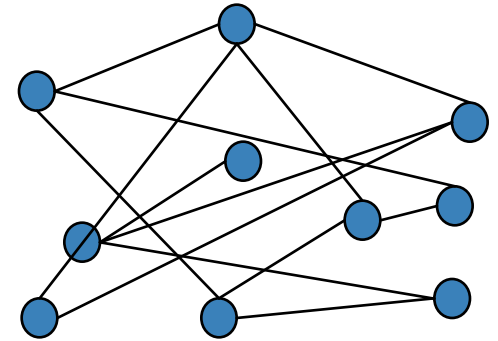
- Modular systems** are systems that are composed of modules
- **Easier to understand**: when dealing with a module the details of other modules can be ignored (separation of concerns)
 - Modules can be **developed & maintained independently**
 - Separation of work: different teams for different modules
 - Independent testing of modules
 - Modules can be **reused** in several systems
 - Modules can be **replaced** by other modules with the same interface
 - **Isolation** between modules can prevent failure in one module to cause failure in other modules



Spaghetti Code vs. Modular System

Spaghetti Code

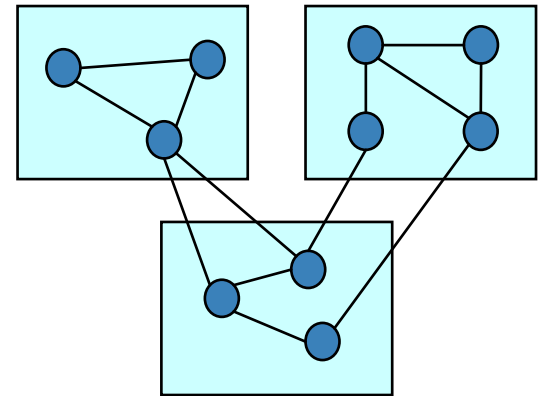
- Haphazard connections, probably grown over time
- No visible cohesive groups
- High coupling: high interaction between random parts
- Understand it all or nothing



10 parts, 13 connections

Modular System

- High cohesion within modules
- Low coupling between modules
- Modules can be understood separately
- Interaction between modules is well-understood and thoroughly specified



10 parts, 13 connections,
3 modules

Information Hiding

Problem: Information Overload

Idea: Hide information that does not need to be visible in order to use a class/module/program

- Too much information can be **confusing**: what is important for usage and what not?
- Too much information can lead to undesired **dependencies**
 - If internals are visible & accessible, someone might use/change them (use something in an unintended manner)
 - If internals are changed then external code that relies on them might not work anymore
- Allowing only restricted access gives us more **flexibility**
 - Class/module/program can be (ex)**changed** without breaking other parts
 - Many design decisions can be hidden and the system design can **evolve without collapsing**



Three-Tier Architecture

Presentation tier

The top-most level of the application is the user interface. The main function of the interface is to translate tasks and results to something the user can understand.



Logic tier

This layer coordinates the application, processes commands, makes logical decisions and evaluations, and performs calculations. It also moves and processes data between the two surrounding layers.



GET LIST OF ALL SALES MADE LAST YEAR



ADD ALL SALES TOGETHER

QUERY

SALE 1
SALE 2
SALE 3
SALE 4

Data tier

Here information is stored and retrieved from a database or file system. The information is then passed back to the logic tier for processing, and then eventually back to the user.



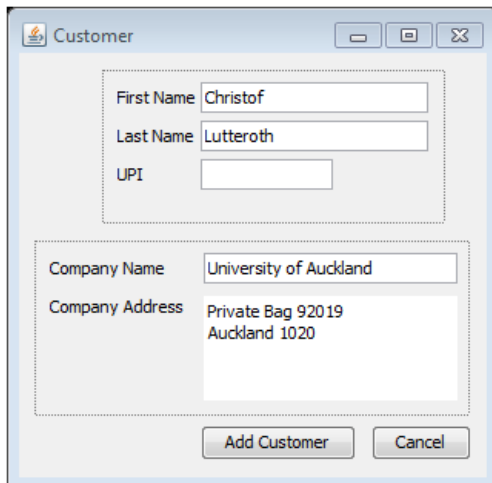
Database

Storage

Separation of concerns

- Tiers can be developed & maintained fairly independently
- Important for system evolution
- Similar to other designs such as model-view-controller (MVC)

Hierarchical Decomposition: Separation of Concerns within a UI



Customer

First Name

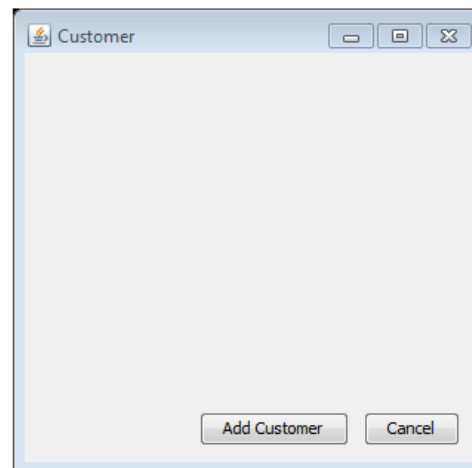
Last Name

UPI

Company Name

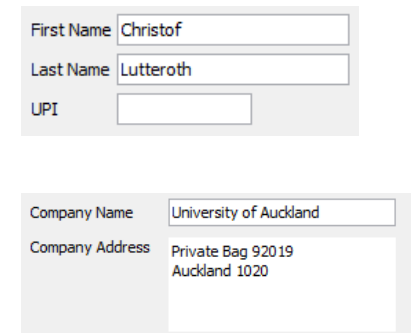
Company Address

=



Customer

+



First Name

Last Name

UPI

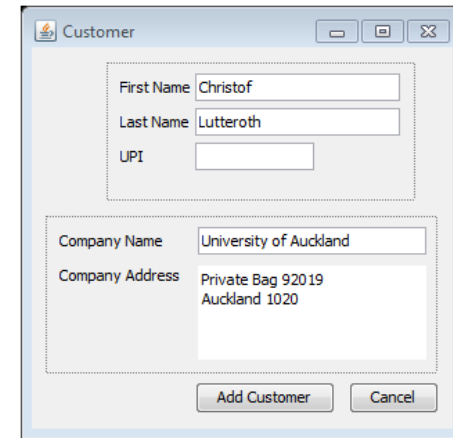
Company Name

Company Address

Hierarchical Decomposition of UIs

Problem: duplication of UI parts, i.e. some UI parts are needed in several places

- Within an application, but also across apps
 - Parts of forms, i.e. fields for entering data, e.g. for personal information
 - Toolbars & menus, e.g. with functions for opening, saving, ...
 - Whole windows and dialogues (e.g. for handling errors)
- Duplication is more work and creates inconsistencies



The image shows a screenshot of a Java Swing window titled "Customer". The window contains two distinct form panels, each enclosed in a dashed border, illustrating the reuse of UI components. The first panel contains three text input fields: "First Name" with the value "Christof", "Last Name" with the value "Lutteroth", and "UPI" which is empty. The second panel contains two text input fields: "Company Name" with the value "University of Auckland" and "Company Address" with the value "Private Bag 92019 Auckland 1020". At the bottom of the window are two buttons: "Add Customer" and "Cancel".

Solution: develop reusable UI parts in separate classes

- Subclass of JPanel to group related widgets
- Subclass of JFrame to create reusable windows
- Reusable parts can themselves reuse other parts...

HIERARCHICAL DECOMPOSITION

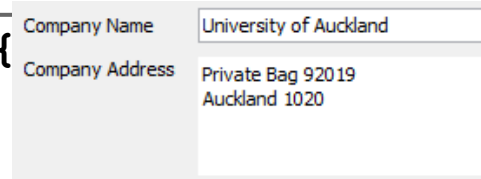
EXAMPLE

```
public class PersonDataForm extends JPanel {  
    private JLabel firstNameLabel;  
    private JTextField firstNameField;  
    private JLabel lastNameLabel; ...  
  
    public PersonDataForm() {  
        firstNameLabel = new JLabel("First Name"); ...  
    }  
}
```



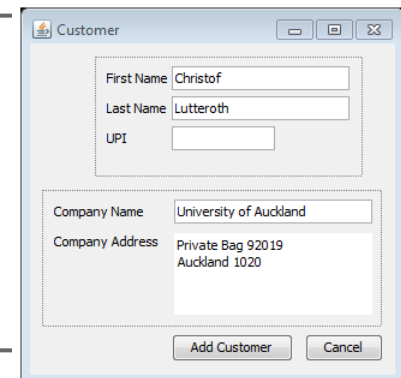
A screenshot of the PersonDataForm UI component. It consists of three input fields: "First Name" with the value "Christof", "Last Name" with the value "Lutteroth", and "UPI" which is currently empty.

```
public class CompanyDataForm extends JPanel {  
    private JLabel companyNameLabel; ...  
}
```



A screenshot of the CompanyDataForm UI component. It consists of two input fields: "Company Name" with the value "University of Auckland" and "Company Address" with the value "Private Bag 92019 Auckland 1020".

```
public class CustomerForm extends JFrame {  
    PersonDataForm personDataForm;  
    CompanyDataForm companyDataForm;  
    JButton addButton; ...  
}
```



A screenshot of the CustomerForm UI component, which is a window titled "Customer". It contains two sub-forms: the PersonDataForm (with "First Name" as "Christof" and "Last Name" as "Lutteroth") and the CompanyDataForm (with "Company Name" as "University of Auckland" and "Company Address" as "Private Bag 92019 Auckland 1020"). At the bottom of the window are two buttons: "Add Customer" and "Cancel".

Separation of UI and Data (Model and View)



represented
in



	A	B	C	D
41	715 S2 C	*	23	3
42	720 S1 C	*	7	3
43	725 S2 C	*	29	3
44	732 S1 C	*	47	3
45	734 S1 T	*	40	3
46	742 S2 C	*	42	3
	750 S2 C	*	8	3

Different Views

SEPARATION OF MODEL AND VIEW

Use different classes for Model and View:

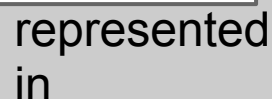
- **Model:** the data that is presented by a widget, i.e. the data storage implementation (classes & methods)
- **View:** the presentation of the data on the screen, i.e. the widgets that paint the data (classes & methods)

The data of a GUI component may be represented using several model objects, e.g. for

- Displayed data (e.g. list items in **JList: ListModel**)
- Widget state (e.g. selections in **JList: ListSelectionModel**)



Data



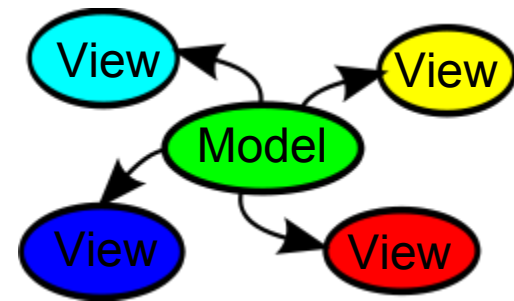
represented in



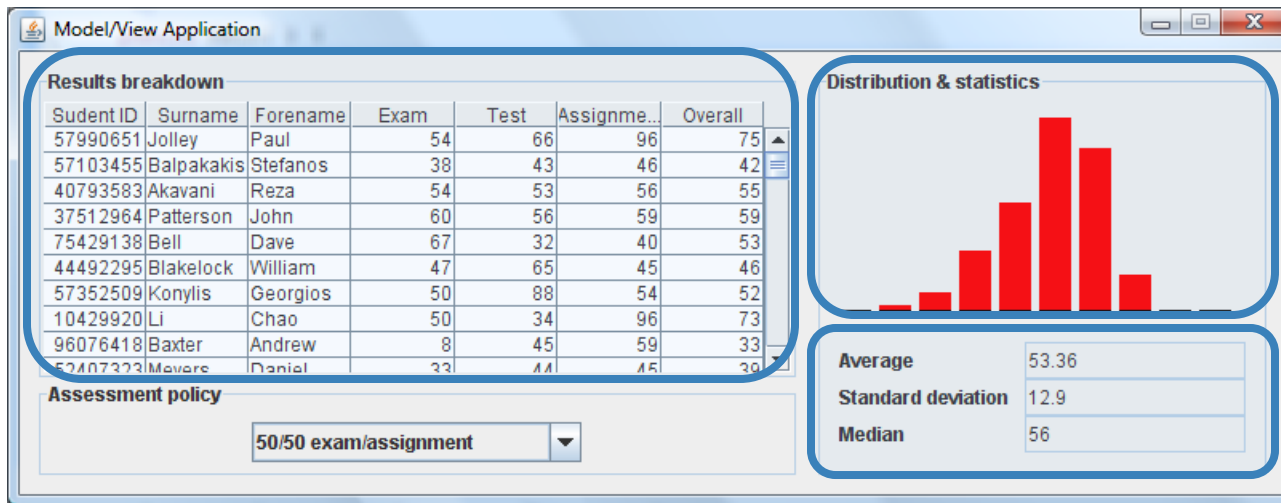
	A	B	C	D
41	715 S2 C	*	23	3
42	720 S1 C	*	7	3
43	725 S2 C	*	29	3
44	732 S1 C	*	47	3
45	734 S1 T	*	40	3
46	742 S2 C	*	42	3
	750 S2 C	*	8	3

Advantages of Model-View SEPARATION

- **Separation of concerns** during development
 - Model can be **developed & maintained independently** from view
 - Well-defined interface between model and view makes sure that they can work together
- New possibilities for connecting models and views
 - Model can be displayed in **multiple views**
 - Models and views can be **distributed**
- Model concept is integrated with **event notification**
 - Changes of the model trigger updates of view
 - Changes of the view trigger updates of model
 - **Consistency** between model and view

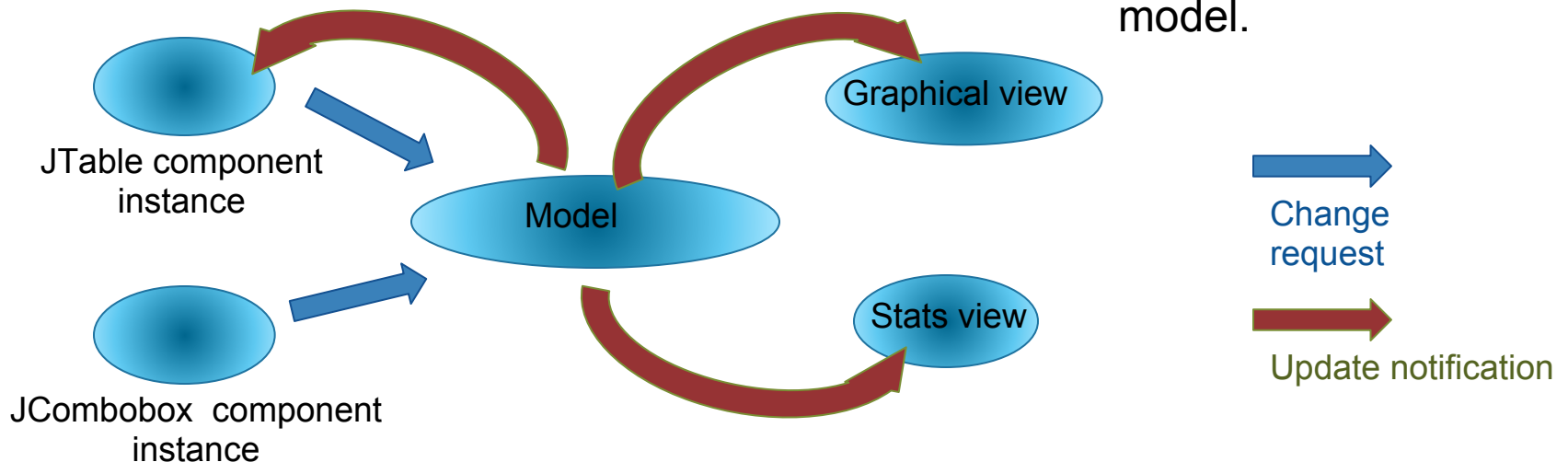


A TYPICAL MODEL-VIEW APPLICATION



Many desktop applications provide multiple **views** of some data model.

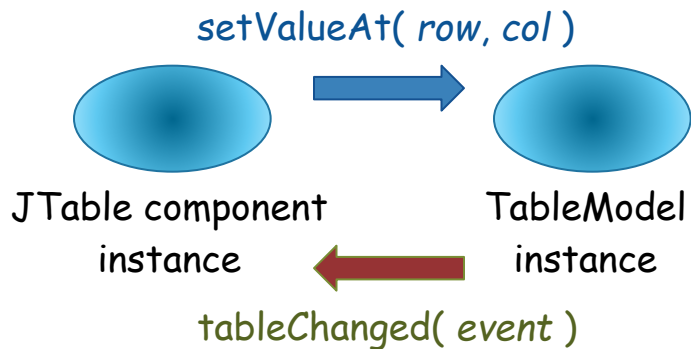
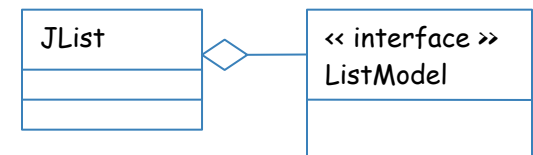
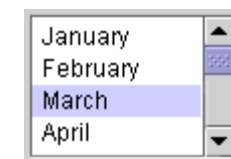
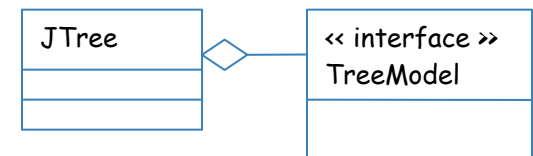
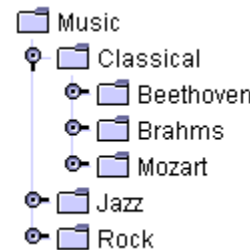
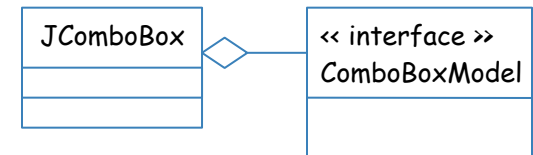
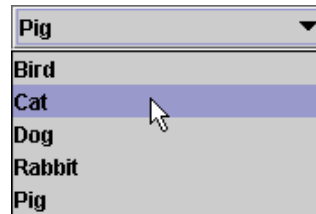
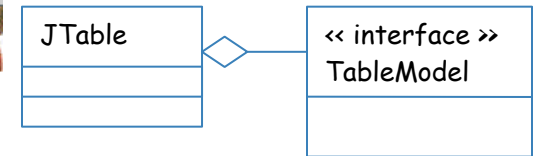
Invariant : all views should offer a mutually consistent representation of the model.



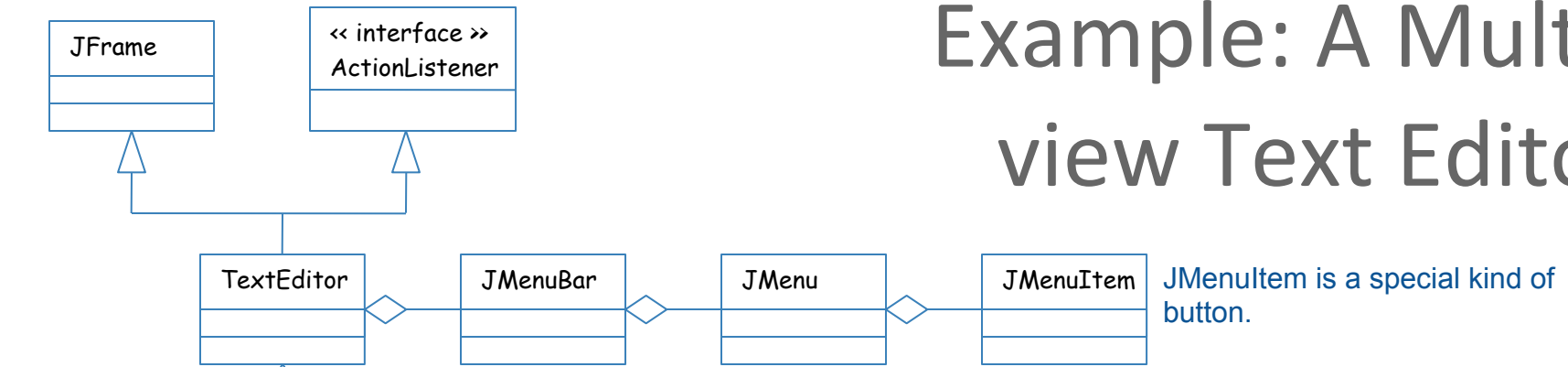
MODEL-VIEW IN SWING

- Contemporary GUI frameworks, like Swing, are based on a **separable model** architecture
- All Swing widgets (JComponents) have separate models

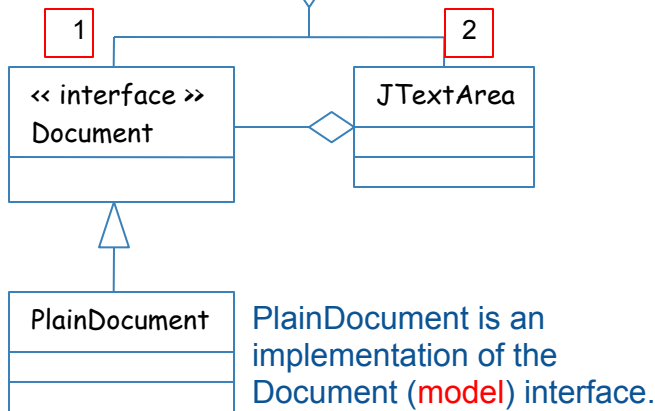
First Name	Last Name	Favorite Food
Jeff	Dinkins	
Ewan	Dinkins	
Amy	Fowler	
Hania	Gajewska	
David	Geary	



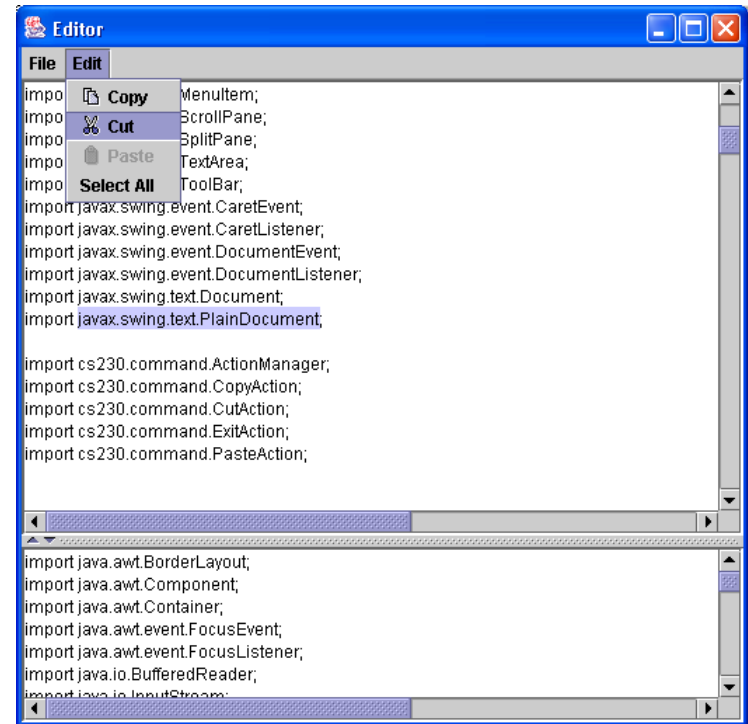
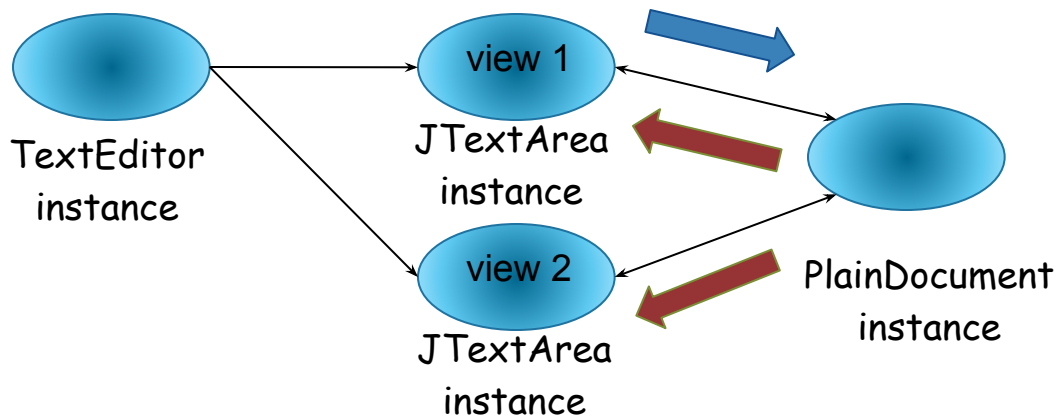
Example: A Multi-view Text Editor



JMenuItem is a special kind of button.



PlainDocument is an implementation of the Document (model) interface.



List Model Example

```
listModel = new DefaultListModel();  
listModel.addElement("Alan Sommerer");  
list = new JList(listModel);  
...  
hireButton.addActionListener(new ActionListener() {  
    void actionPerformed(ActionEvent e) {  
        listModel.addElement(nameField.getText());  
    }  
});  
  
fireButton.addActionListener(new ActionListener() {  
    void actionPerformed(ActionEvent e) {  
        int index = list.getSelectedIndex();  
        listModel.remove(index);  
    }  
});  
...
```



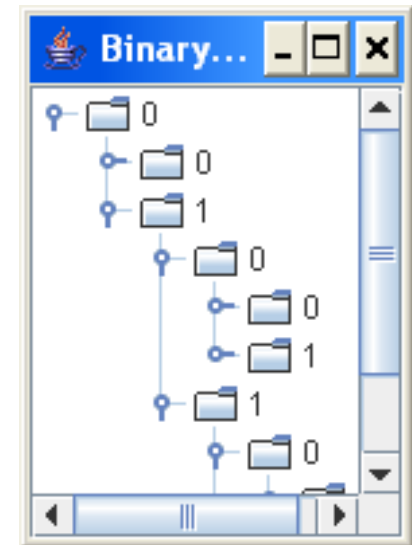
Full source code at:

<http://docs.oracle.com/javase/tutorial/uiswing/components/list.html>

Tree Model Example Part 1

```
import javax.swing.tree.*;
import javax.swing.event.*;
...
public class BinaryTree implements TreeModel {
    public Object getRoot() { return 0; }
    public int getChildCount(Object parent) { return 2; }
    public Object getChild(Object parent, int index) {
        return index;
    }
    public int getIndexOfChild(Object parent, Object child) {
        return (Integer)child;
    }
    public boolean isLeaf(Object node) {
        return false;
    }

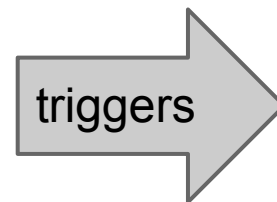
    // see next slide for more...
}
```



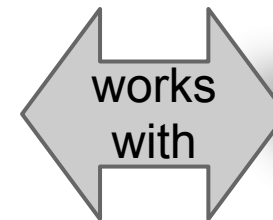


Separation of UI (View) and Application Logic

	A	B	C	D
41	715 S2 C	*	23	3
42	720 S1 C	*	7	3
43	725 S2 C	*	29	3
44	732 S1 C	*	47	3
45	734 S1 T	*	40	3
46	742 S2 C	*	42	3
47	750 S2 C	*	8	3



Logic



Data

SEPARATION OF View and Logic

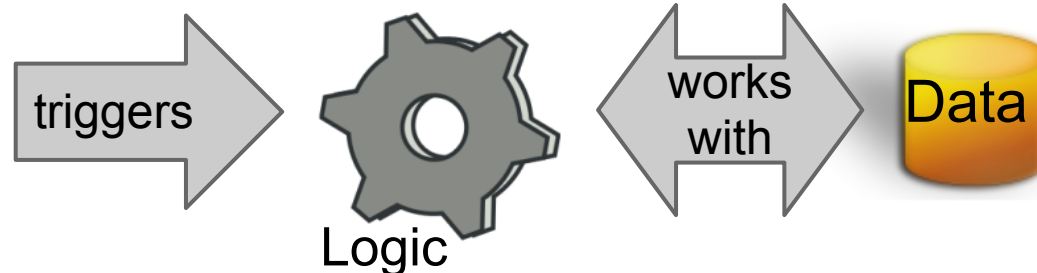
Use different classes for View and Logic:

- **View:** the presentation of the data on the screen, i.e. the widgets that paint the data (classes & methods)
- **Logic:** the operations that the program performs, e.g. decisions, calculations, data processing/filtering, etc.

The logic of an application is implemented in **your own classes**

- **Methods** for the different operations triggered through the UI that read data from the model and work with it
- Should have a well-defined **interface** to view
- Main advantage: easier development & **maintenance** through separation of concerns

	A	B	C	D
41	715 S2 C	*	23	3
42	720 S1 C	*	7	3
43	725 S2 C	*	29	3
44	732 S1 C	*	47	3
45	734 S1 T	*	40	3
46	742 S2 C	*	42	3
47	750 S2 C	*	8	3



Separation of Logic Example

```
...  
hireButton.addActionListener(new ActionListener() {  
    void actionPerformed(ActionEvent e) {  
        String name = nameField.getText();  
        logic.hire(name);  
    }});  
  
fireButton.addActionListener(new ActionListener() {  
    void actionPerformed(ActionEvent e) {  
        String name = (String) list.getSelectedValue();  
        logic.fire(name);  
    }});  
...
```

Logic class defines methods for hiring and firing
e.g. `hire()`

- **Validate input:** check if the name is correct
- **Check data constraints:** make sure there is a vacancy
- **Update model:** add new employee



Summary



- **Separation of Concerns, Modularity** and **Information hiding** are important design principles
- Improved reuse & maintenance through
 - Hierarchical Decomposition
 - Separation of Model and View
 - Separation of View and Logic

Assignment 3 out today:

Design and implement your own GUI prototype

Test this Friday during the lecture time:

From Christof's part only week 7 covered (first week)

Quiz



1. Briefly describe the three tiers of a 3-tier architecture.
2. What does separation of model and view mean?
Describe two of the advantages.
3. Why is it good to separate the logic and the view of an application?



USB Coffee Machine <http://vivifyer.deviantart.com/art/USB-Coffee-Machine-56399525>