SOFTWARE ENGINEERING

Design Components

Johannes Sametinger
Institut für *WirtschaftsINformatik*
SOFTWARE ENGINEERING
Johannes Kepler Universität Linz, Austria
Universität Regensburg, Germany

Problems and Goals

SOFTWARE ENGINEERING

- design expertise
  - » design expertise not available in composable form
- design blurring
  - » design blurring and degradation during subsequent modifications
- documentation
  - » documentation of designs and design decisions
- documentation
  - » characteristics of good/bad designs

→ package software architectures that are based on well-defined design expertise into tangible, customizable, and composable design components

→ provide addtional design view on software system

→ provide design documentation

Outline



- Design Patterns
- Design Components
- Design Roles
- Component Models
- Component Types
- Example
- Design Constraints
- Documentation
- Tool Support
- Related Work
- Conclusion

Causes of Redesign



- creating objects by specifying a class explicitly
- dependence on
  - » specific operations
  - » hardware and software platform
  - » object representations or implementations
- algorithmic dependencies
- tight coupling
- extending functionality by subclassing
- inability to alter classes conveniently

Design Patterns

SOFTWARE ENGINEERING

Christopher Alexander:

- » Each pattern describes a problem which occurs over and over again in our environment, and then
- » describes the core of the solution to that problem,
- » in such a way that you can use this solution a million times over,
- » without ever doing it the same way twice.

Gamma, Helm, Johnson, Vlissides:

- » A design pattern systematically names, motivates, and explains a general design that addresses a recurring design problem in object-oriented systems.
- » It describes the problem, the solution, when to apply the solution, and its consequences.
- » It also gives implementation hints and examples. The solution is a general arrangement of objects and classes that solve the problem. The solution is customized and implemented to solve the problem in a particular context.

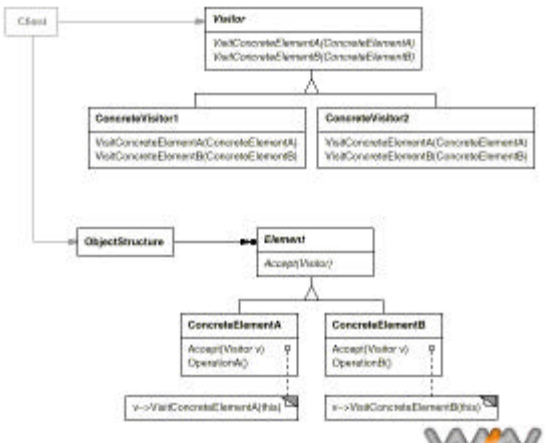
Design Patterns II

SOFTWARE ENGINEERING

Gamma, Helm, Johnson, Vlissides

- » Name and Classification
- » Intent
- » Also Known As
- » Motivation
- » Applicability
- » Structure
- » Participants
- » Collaborations
- » Consequences
- » Implementation
- » Sample Code
- » Known Uses
- » Related Patterns

[Gamma et al. 95]



What to Expect from a Design Pattern



- common design vocabulary
- documentation and learning aid
- adjunct to existing methods
- target for refactoring

How to Use a Design Pattern



- read the pattern once through for an overview
- go back and study the Structure, Participants, and Collaborations sections
- look at the Sample Code section to see a concrete example of the pattern in code
- choose names for pattern participants that are meaningful in the application context
- define the classes
- define application-specific names for operations in the pattern
- implement the operations to carry out the responsibilities and collaborations in the pattern

Design Components — Motivation

SOFTWARE ENGINEERING

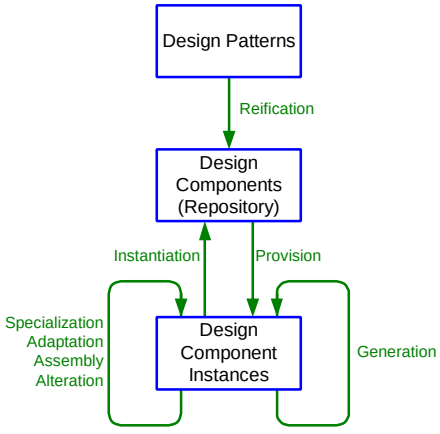
- impediments to pattern-based software engineering
  - » design patterns are treated only as non-software artifacts;
  - » programmers create, extend, and modify classes throughout the software;
  - » major maintenance problem: programmers tend to lose sight of the original patterns
- design components
  - » address this blurring of design patterns during implementation and maintenance, and
  - » suggest a more systematic approach to define, implement, and trace them within a component-based development cycle.

[Keller/Schauer 98]

Design Composition Process

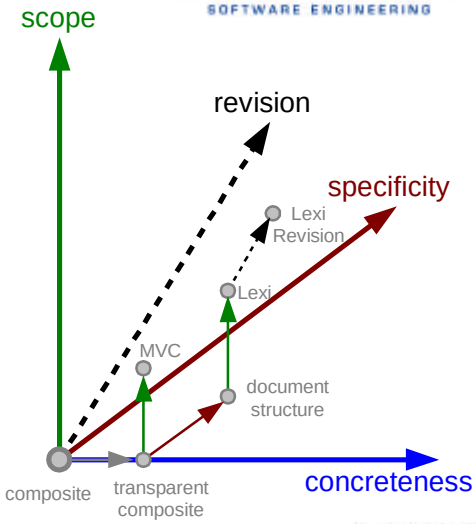
SOFTWARE ENGINEERING

- design components
  =
  reified design patterns
- design composition by
  - » instantiation
  - » specialization
  - » alteration
  - » adaptation
  - » assembly
  - » provision
  - » generation



Dimensions of Design Composition

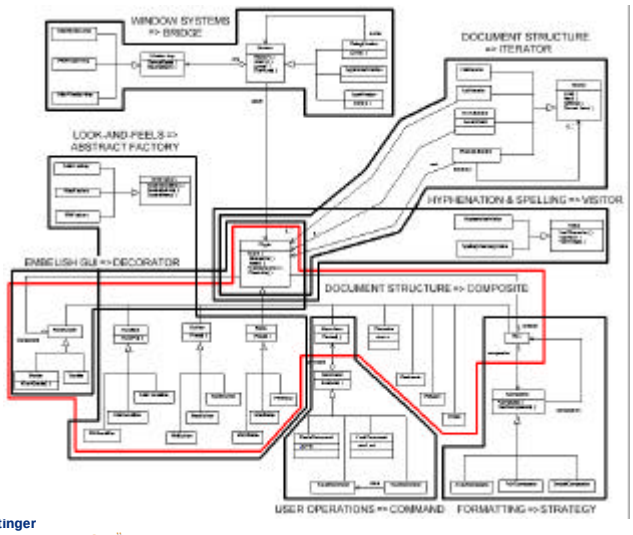
- concreteness
  abstract → concrete
- specificity
  generic → domain specific
- scope
  basic → composite
- revision
  original → current



Johannes Sametinger
JOHANNES KEPLER UNIVERSITÄT LINZ
UNIVERSITÄT REGENSBURG

INSTITUT FÜR WIRTSCHAFTSINFORMATIK

Design Components — Example



Johannes Sametinger
JOHANNES KEPLER UNIVERSITÄT LINZ
UNIVERSITÄT REGENSBURG

INSTITUT FÜR WIRTSCHAFTSINFORMATIK

Object Modeling



- objects
  - » runtime entities that provide state and operations to query and change that state
- classes
  - » definitions of sets of objects (instances), specifying structure and behavior of its instances
- inheritance
- object associations
  - » pairs of objects of which one holds a reference to the other
- object aggregations
  - » pairs of objects of which one is part of the other
- object collaborations
  - » sets of objects that relate to each other by object relationships

Role Modeling



- roles
  - » observable behavioral aspects of an object

  Objects, which provide a particular role, are said to *play* that role.
- object collaboration tasks
  - » object collaborations and set of roles objects play in a collaboration
- role types
  - » types that define the behavior of roles objects may play
  - » define the operations and the state model of roles, as well as the associated semantics
- role models
  - » sets of role types that relate to each other

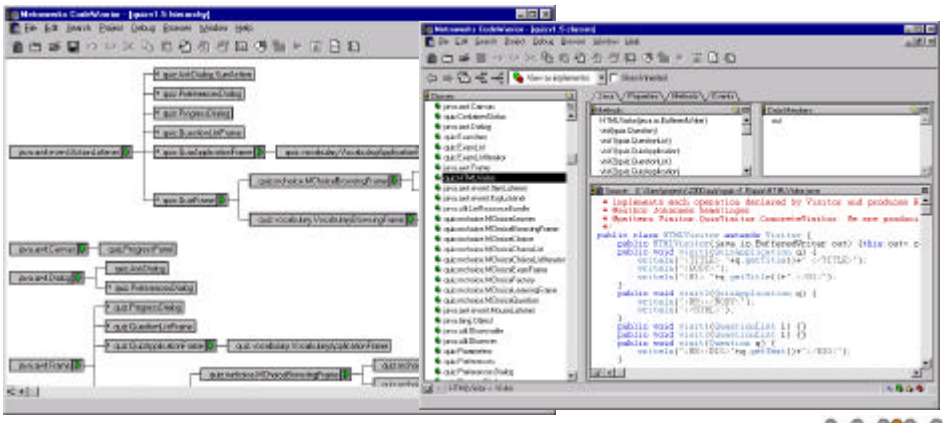
[Riehle 2000]

Views on Software Systems

SOFTWARE ENGINEERING

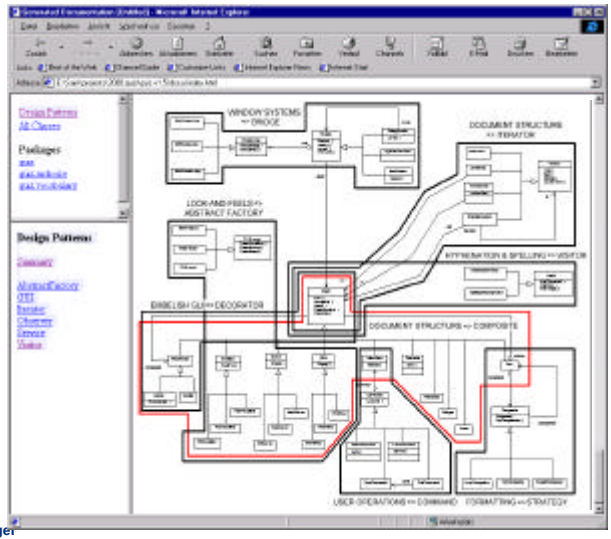
– hierarchy browser

– class browser



Design View

SOFTWARE ENGINEERING

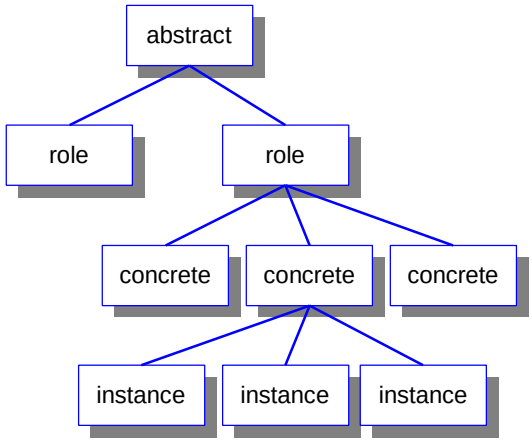


Design Component Models

SOFTWARE ENGINEERING

- abstract
- role
- concrete
- instantiation

- parameterization
- model hierarchy



Design Component Models: 1. Abstract Model

SOFTWARE ENGINEERING

Visitor

- Intent
  - » Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.
- Motivation
  - » Consider a compiler that represents programs as abstract syntax trees. It will need to perform operations on abstract syntax trees for "static semantic" analyses like checking that all ...
- Applicability
  - » Use the Visitor pattern when
    - an object structure contains many classes of objects with differing interfaces, and you want to perform operations on these objects that depend on their concrete classes.
    - many distinct and unrelated operations need to be performed on objects in an object structure, and you want to avoid "polluting" their classes with these operations. Visitor lets you keep ...

Design Component Models: 2. Role Model

SOFTWARE ENGINEERING

- AbstractVisitor
  - » declares a Visit operation for each ConcreteElement in the object structure.
- ConcreteVisitor
  - » implements each operation declared by Visitor.
  - » provides the context for the algorithm and stores its local state.
- AbstractElement
  - » defines an Accept operation that takes a visitor as an argument.
- ConcreteElement
  - » implements an Accept operation that takes a visitor as an argument.
- ObjectStructure
  - » can enumerate its elements.
  - » may either be a composite or a collection such as a list or a set.

Design Component Models: 3. Concrete Model

SOFTWARE ENGINEERING

```
class Visitor {
public:
   virtual void VisitElementA(ElementA*);
   virtual void VisitElementB(ElementB*);
// and so on for other concrete elements
protected:
   Visitor();
};
class Element {
public:
   virtual ~Element();
   virtual void Accept(Visitor&) = 0;
protected:
   Element();
};
```

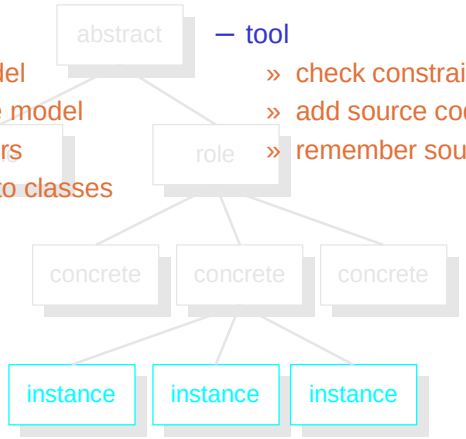
```
class CompositeElement : public Element {
public:
   virtual void Accept(Visitor&);
private:
   List<Element*>* _children;
};

void CompositeElement::Accept (Visitor& v) {
   ListIterator<Element*> i(_children);
   for (i.First(); !i.IsDone(); i.Next()) {
      i.CurrentItem()->Accept(v);
   }
   v.VisitCompositeElement(this);
}
```

Design Component Models: 4. Instantiation



- user
  - » pick role model
  - » pick concrete model
  - » set parameters
  - » assign roles to classes
  - » customize
- tool
  - » check constraints
  - » add source code
  - » remember source code



Abstract Visitor Model




```
abstractmodel {
    name:          Visitor Component
    author:        Johannes Sametinger
    date:          7/27/2000
    version:       v0.0
    short:         design component based on Visitor Pattern by Gamma et al.
    intent:        Represent an operation to be performed on the elements of ...
    motivation:    Consider a compiler that represents programs as abstract  ...
    applicability: An object structure contains many classes of objects with ...
                   Many distinct and unrelated operations need to be...
                   The classes defining the object structure rarely change, ...
    consequence:   Visitor makes adding new operations easy. ...
                   A visitor gathers related operations and separates ...
                   Visiting across class hierarchies. ...
                   Accumulating state. ...
    knownuse:      The Smalltalk-80 compiler has a Visitor class ...
                   IRIS Inventor is a toolkit for developing 3-D graphics ...
                   To make adding new nodes easier, Inventor implements ...
}
```

Visitor Role Model




```
rolemodel {
    abstract:       Visitor Component
    name:           default
    description:    A client that uses the Visitor pattern must create...
    roletype AbstractVisitor:       declares a Visit operation for each class of ...
    roletype ConcreteVisitor:       implements each operation ...
    roletype AbstractElement:       defines an Accept operation that takes a visitor ...
    roletype ConcreteElement:       implements an Accept operation that ...
    roletype ObjectStructure:       can enumerate its elements, may ...
}
```


Concrete Visitor Model




```
concrete {
    abstract:       Visitor Component
    rolemodel:      default
    platform:       Java2
    roletype AbstractVisitor
        abstract public void visit($ConcreteElement$ e);
    roletype ConcreteVisitor
        public void visit($ConcreteElement$ e) { #anyCode# }
    roletype AbstractElement
        abstract public void accept($AbstractVisitor$ v);
    roletype ConcreteElement
        public void accept($AbstractVisitor$ v) { v.visit$ConcreteElement$ (this); }
    roletype ObjectStructure
        void #doAnything#() {accept(new $ConcreteVisitor$());}
}
```

Visitor Instantiation




```
package quiz;
/** represents an operation to be performed on the elements of a quiz
 * @author          Johannes Sametinger
 * @pattern         Visitor.AbstractVisitor.QuizVisitor; declares a visit operation
 *                  for each class in the quiz (ConcreteElement)
 */
abstract public class Visitor {
    public Visitor() {}
    /** operation to be performed on objects of class Question
     * @pattern      Visitor.Visitor.QuizVisitor  visit operation for class Question
     */
    abstract public void visit(Question q);
    /** operation to be performed on objects of class QuestionList
     * @pattern      Visitor.Visitor.QuizVisitor  visit operation for class QuestionList
     */
    abstract public void visit(QuestionList l);
}
```

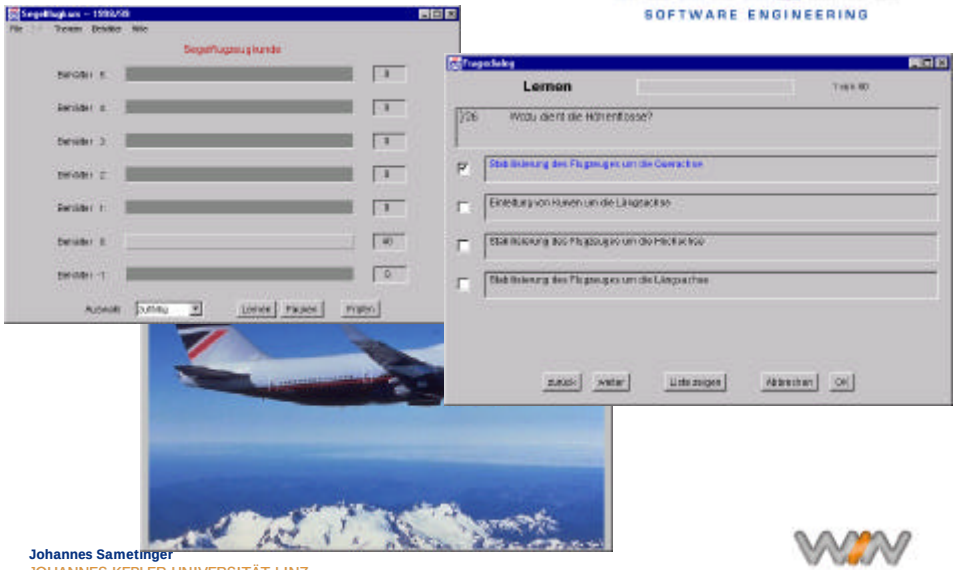

Parameters




```
concrete {
    abstract:       Visitor Component
    rolemodel:      default
    platform:       Java2
    roletype AbstractVisitor
            alternative single call
                    abstract public void visit($ConcreteElement$ e);
            alternative double call
                    abstract public void visit1$ConcreteElement$ e);
                    abstract public void visit2($ConcreteElement$ e);
    roletype ConcreteVisitor
            alternative single call
                    public void visit($ConcreteElement$ e) { #anyCode# }
            alternative double call
                    public void visit1($ConcreteElement$ e) { #anyCode# }
                    public void visit2($ConcreteElement$ e) { #anyCode# }
    ...
}
```

Example — Quiz Application

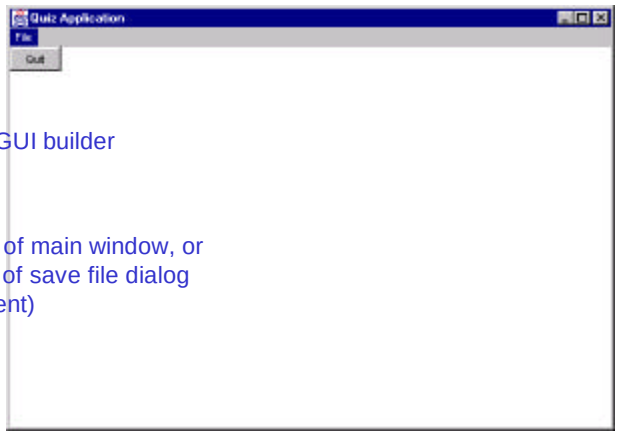
SOFTWARE ENGINEERING



Quiz V0.0—empty application

SOFTWARE ENGINEERING

GUI component #1



- GUI role
  typically created by a GUI builder
- client role
  main method as client of main window, or
  main window as client of save file dialog
  (another GUI component)

Quiz V0.0 — GUI Component Roles



— GUI role

```
public class QuizApplFrame extends java.awt.Frame implements
      java.awt.event.ActionListener {
      public QuizApplFrame () {
            super("Quiz Application");
            setSize(new java.awt.Dimension(600,400));
            setupMenus();
      }
      java.awt.MenuBar mb;
      java.awt.Menu fileMenu;
      java.awt.MenuItem quitMenuItem;
      private void setupMenus() {
            quitMenuItem= new java.awt.MenuItem("Quit");
            quitMenuItem.addActionListener(this);
            fileMenu= new java.awt.Menu("File");
            fileMenu.add(quitMenuItem);
            mb = new java.awt.MenuBar();
            mb.add(fileMenu);
            setMenuBar(mb);
      }
      public void actionPerformed(java.awt.event.ActionEvent e) {
            if (e.getSource()==quitMenuItem) quit();
      }
      public boolean quit() {
            dispose(); System.exit(0); return true;
      }
}
```

— client role

```
public class QuizMain {
      public static void main(String[] args) {
            QuizApplFrame app= new QuizApplFrame();
            app.setVisible(true);
      }
}
```

— glue
attach system functionality to user activities

— customization
access application-specific data to customize user interface, e.g., dimming menu items

Quiz V0.1 — Model Components



— client

```
class QuizApplFrame

      private QuizApplication quizApplication;
      public void setQuizApplication(QuizApplication q)
            {quizApplication= q;}
      public QuizApplication getQuizApplication()
            {return quizApplication;}
```

— model

```
public class QuizApplication extends Object {
      public QuizApplication() {}
```

— client

```
      private QuestionList questionList;
      public void setQuestionList(QuestionList questionList)
            {this.questionList= questionList;}
      public QuestionList getQuestionList()
            {return questionList;}
}
```

— model

```
public class QuestionList extends Object
{
      public QuestionList() {}
      private java.util.Vector list= new java.util.Vector();
      public void add(Question q) {list.addElement(q);}
      public void remove(Question q) {list.removeElement(q);}
      public int getSize() {return list.size();}
      public Question getElementAt(int index) {
            try {return (Question)list.elementAt(index);}
            catch(Exception e) {return null;}
      }
}
```

— data of application
— attributes
  » e.g., text of question, texts of multiple choices, correct choice

Quiz V0.2 — Service Components

SOFTWARE ENGINEERING

— client roles

class **QuizApplFrame**

```
public void actionPerformed (java.awt.event.ActionEvent e) {
if (e.getSource()==quitMenuItem)
      {quit();return;}
if (e.getSource()==openMenuItem)
      {open();return;}
if (e.getSource()==saveMenuItem)
      {save();return;}
if (e.getSource()==questionMenuItem)
      {question();return;}
...
}
```

- one primary, several secondary roles, at least one client
- primary role is played by whoever starts a service
- participants contributing to service play a secondary role
- typical: scattered source code

— primary role

class **QuizApplFrame**

```
java.awt.FileDialog saveFileDialog= null;
String saveDirectory= null;
String saveFileName= null;
public void save() {
    if (saveFileDialog==null) {
        saveFileDialog= new java.awt.FileDialog(this,"save file");
        saveFileDialog.setMode(java.awt.FileDialog.SAVE);
        saveFileDialog.setSize(new java.awt.Dimension(650,450));
    }
    if (saveDirectory!=null) saveFileDialog.setDirectory(saveDirectory);
    saveFileDialog.setVisible(true);
    saveDirectory= saveFileDialog.getDirectory();
    saveFileName= saveFileDialog.getFile();
    if (saveFileName==null) return;
    java.awt.Cursor c= getCursor();
    setCursor(new java.awt.Cursor(java.awt.Cursor.WAIT_CURSOR));
    quizApplication.save(saveDirectory,saveFileName);
    setCursor(c);
}
```

another GUI client

Johannes Sametinger
JOHANNES KEPLER UNIVERSITÄT LINZ
UNIVERSITÄT REGENSBURG

INSTITUT FÜR WIRTSCHAFTSINFORMATIK

Quiz V0.2 — Service Components II

SOFTWARE ENGINEERING

class **QuizApplication**

```
public boolean save(String directoryName, String fileName) {
    try {
        java.io.BufferedWriter out = new ...;
        if (out==null) return false;
        write(out);
        out.close();
    } catch (java.io.IOException e) {
        return false;
    }
    return true;
}

protected void write(java.io.BufferedWriter out)
    throws java.io.IOException
{
    if (questionList==null) return;
    questionList.write(out);
}
```

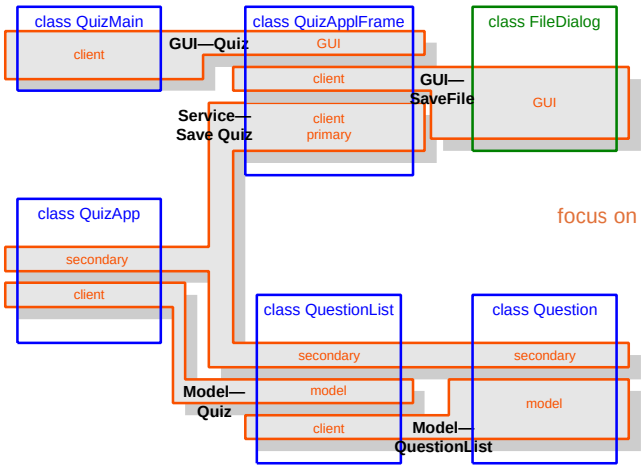
class **QuestionList**

```
protected void write(java.io.BufferedWriter out)
    throws java.io.IOException
{
    QuestionListIterator iter= getIterator();
    Question q= iter.getFirst();
    while (q!=null) {
        q. write(out);
        out.newLine();
        q= iter.getNext();
    }
}
```

Johannes Sametinger
JOHANNES KEPLER UNIVERSITÄT LINZ
UNIVERSITÄT REGENSBURG

INSTITUT FÜR WIRTSCHAFTSINFORMATIK

Quiz V0.2 — Overview

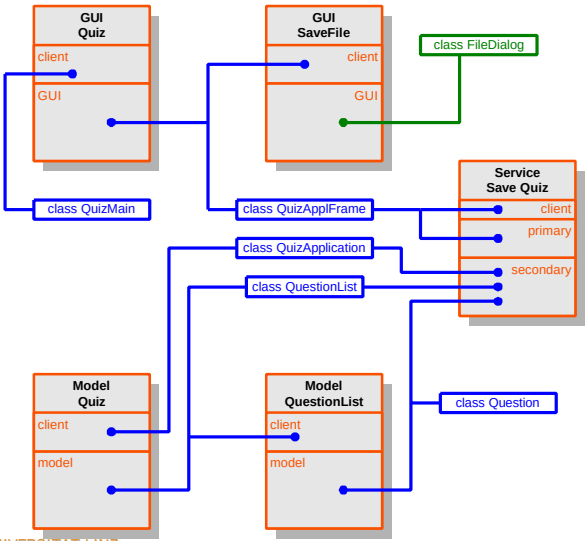


focus on classes

Johannes Sametinger
JOHANNES KEPLER UNIVERSITÄT LINZ
UNIVERSITÄT REGENSBURG

INSTITUT FÜR WIRTSCHAFTSINFORMATIK

Quiz V0.2 — Design Overview



focus on
design components

Johannes Sametinger
JOHANNES KEPLER UNIVERSITÄT LINZ
UNIVERSITÄT REGENSBURG

INSTITUT FÜR WIRTSCHAFTSINFORMATIK

Iterator

SOFTWARE ENGINEERING

- **Intent**
  Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

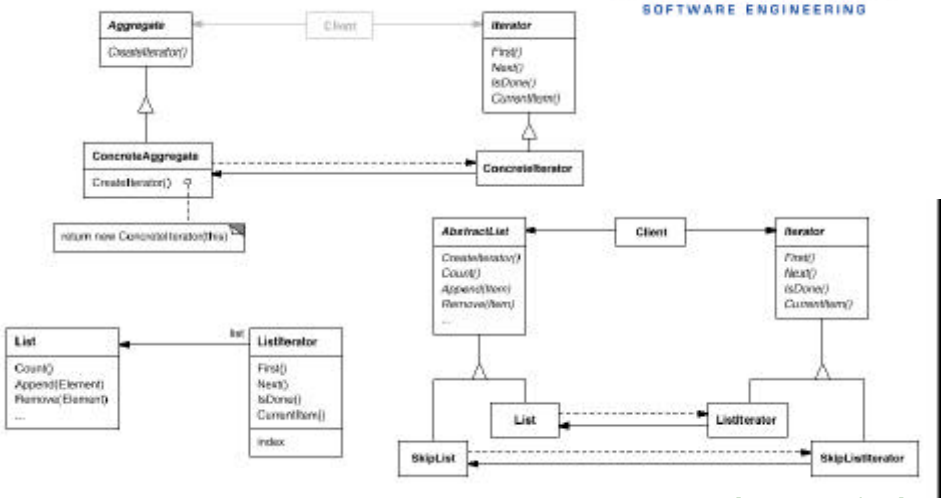
- **Motivation**
  An aggregate object such as a list should give you a way to access its elements without exposing its internal structure. Moreover, you might want to traverse the list in different ways, depending on what you want to accomplish. But you probably don't want to bloat the List interface with operations for different traversals, even if you could anticipate the ones you will need. You might also need to have more than one traversal pending on the same list.
  ...

Participants

- **Iterator**
  defines an interface for accessing and traversing elements.
- **ConcreteIterator**
  implements the Iterator interface, keeps track of the current position in the traversal of the aggregate.
- **Aggregate**
  defines an interface for creating an Iterator object.
- **ConcreteAggregate**
  implements the Iterator creation interface to return an instance of the proper ConcreteIterator.

Iterator II

SOFTWARE ENGINEERING



[Gamma et al. 95]

Iterator Component — Role Assignment

SOFTWARE ENGINEERING

— existing classes

List

```
class QuestionList
    protected void write(java.io.BufferedWriter out)
        throws java.io.IOException
    {
        QuestionListIterator iter= getIterator();
        Question q= iter.getFirst();
        while (q!=null) {
            q. write(out);
            out.newLine();
            q= iter.getNext();
        }
    }
```

Client

```
abstract public class Question {
    public Question() {}
    private String text;
    public void setText(String text) {this.text= text;}
    public String getText() {return text;}
    abstract protected void read(java.io.BufferedReader in)
        throws java.io.IOException;
    abstract protected void write(java.io.BufferedWriter out)
        throws java.io.IOException;
}
```

Item

— new class

```
public class QuestionListIterator extends Object {
    private QuestionList list;
    private int index;
    public QuestionListIterator(QuestionList l) {list= l;}
    public void reset() {index= 0;}
    public Question getFirst() {reset(); return getNext();}
    public Question getNext() {
        if (list==null) return null;
        return list.getElementAt(index++);
    }
    public Question getCurrent() {
        if (list==null) return null;
        return list.getElementAt(index-1);
    }
    public boolean isDone() {
        if (list==null) return true;
        return list.getSize()<=index;
    }
}
```

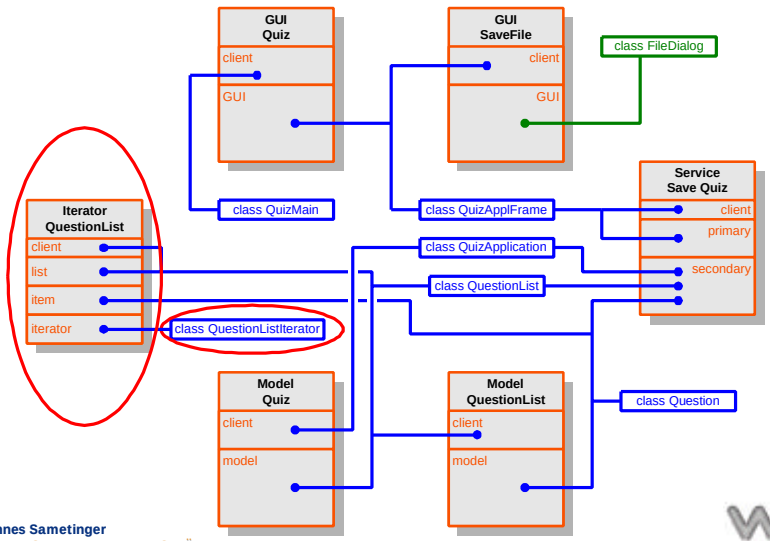
Iterator

Johannes Sametinger
JOHANNES KEPLER UNIVERSITÄT LINZ
UNIVERSITÄT REGENSBURG

INSTITUT FÜR WIRTSCHAFTSINFORMATIK

Quiz V0.3 — Design View

SOFTWARE ENGINEERING

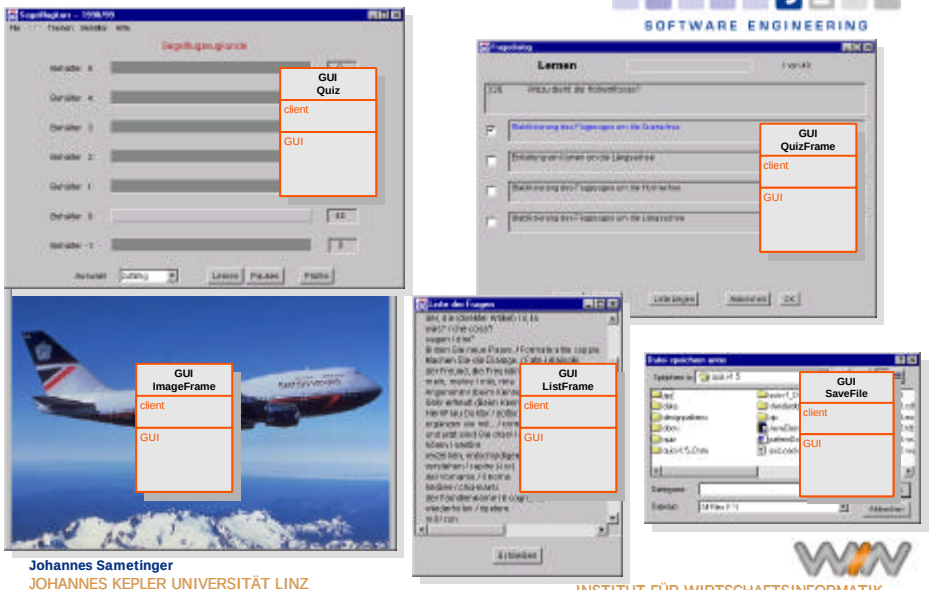


Johannes Sametinger
JOHANNES KEPLER UNIVERSITÄT LINZ
UNIVERSITÄT REGENSBURG

INSTITUT FÜR WIRTSCHAFTSINFORMATIK

More GUI Components

SOFTWARE ENGINEERING



Johannes Sametinger
JOHANNES KEPLER UNIVERSITÄT LINZ
UNIVERSITÄT REGENSBURG

Adding Flexibility — Abstract Factory

SOFTWARE ENGINEERING

- Various forms of questions, e.g., single choice questions, multiple choice questions, text questions, boolean questions

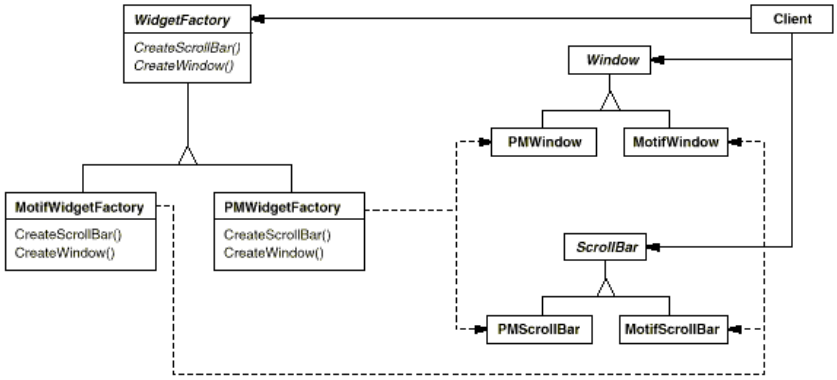
Vocabulary
hoax
?

Multiple Choice
When will an aircraft wing always stall?
a. The indicated airspeed is below the power off stall speed.
b. The angle of attack is greater than the stall angle of attack.
c. The calibrated airspeed is below the power off stall speed.
d. The pitch attitude is nose-up.

- Various tests
  e.g., driver's license, pilot license
- Abstract Factory
  Intent:
  Provide an interface for creating families of related or dependent objects without specifying their concrete classes.

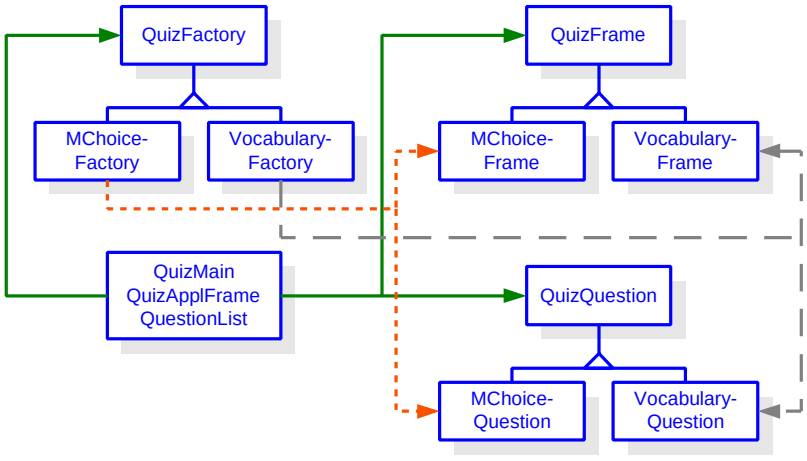
Johannes Sametinger
JOHANNES KEPLER UNIVERSITÄT LINZ
UNIVERSITÄT REGENSBURG

Abstract Factory — Structure



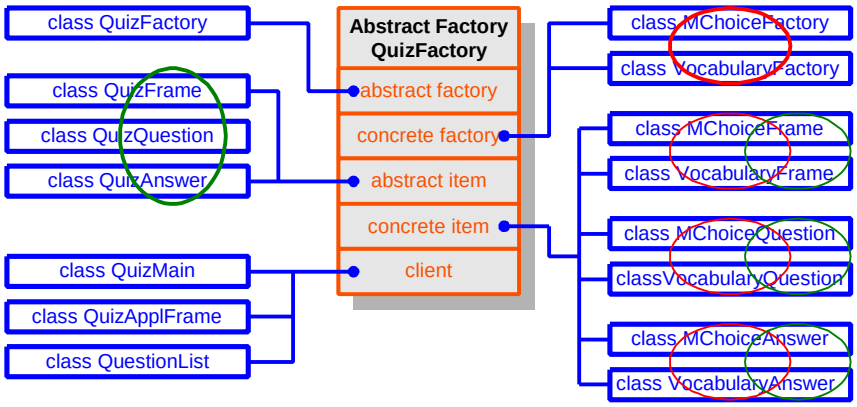
[Gamma et al. 95]

QuizFactory Classes



QuizFactory — Design View

SOFTWARE ENGINEERING

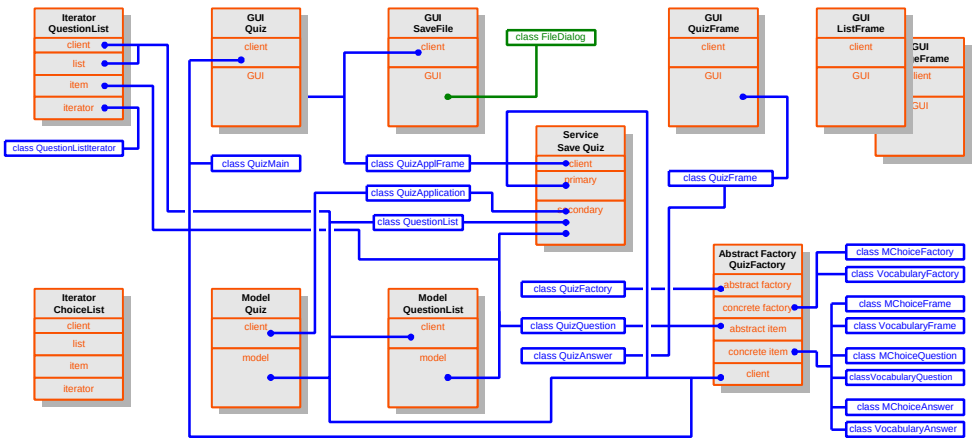


Johannes Sametinger
JOHANNES KEPLER UNIVERSITÄT LINZ
UNIVERSITÄT REGENSBURG

INSTITUT FÜR WIRTSCHAFTSINFORMATIK

Quiz V0.4 — Design View

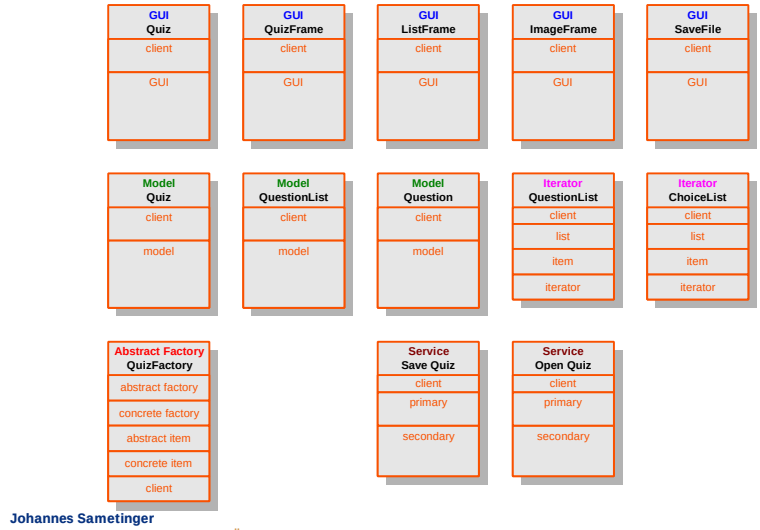
SOFTWARE ENGINEERING



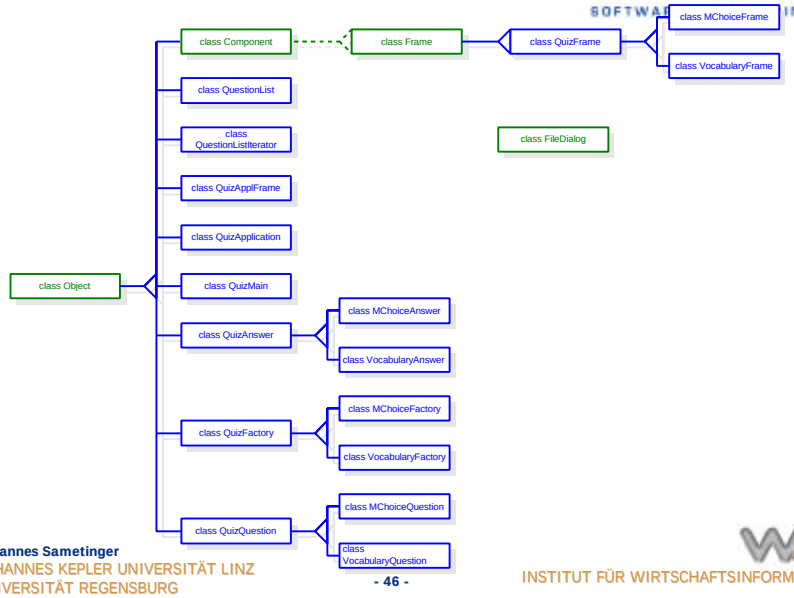
Johannes Sametinger
JOHANNES KEPLER UNIVERSITÄT LINZ
UNIVERSITÄT REGENSBURG

INSTITUT FÜR WIRTSCHAFTSINFORMATIK

Quiz V0.4 — Design View II



Quiz V0.4 — Inheritance Tree



Component Types



- model components
  - » capture an application's model
  - » define attributes
- GUI components
  - » graphical user interface
  - » e.g., dialog, window
- architecture components
  - » architectural structure
  - » e.g., pipes/filters, event-based systems, layered systems
- service components
  - » parts of the system that logically belong together, but are spread all over the system
  - » e.g., save data structure
- pattern components
  - » reification of design expertise
  - » e.g., abstract factory, iterator, visitor

Quiz Application — Design Components

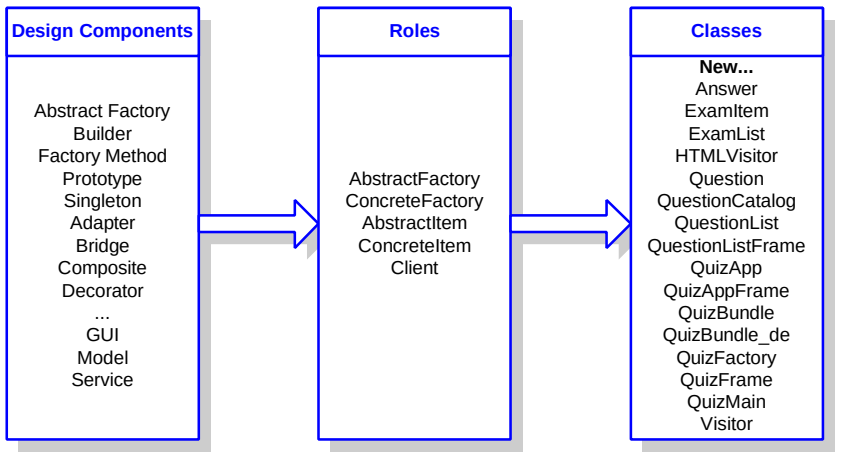


- model components
  - » e.g., quiz, question, choice, exam
- GUI components
  - » quiz, question, image, open/save
  - » internationalization
- service components
  - » text I/O, spread over quiz, question, choice
- pattern components
  - » iterator, visitor, observer, abstract factory

Source code:	
# LOC	~2.000
# classes	35
# interfaces	2
# GUI classes	10
# packages	2
Design components:	
# total	33
# GUI	10
# model	8
# service	3
# abstract factory	1
# iterator	5
# observer	3
# template method	2
# visitor	1

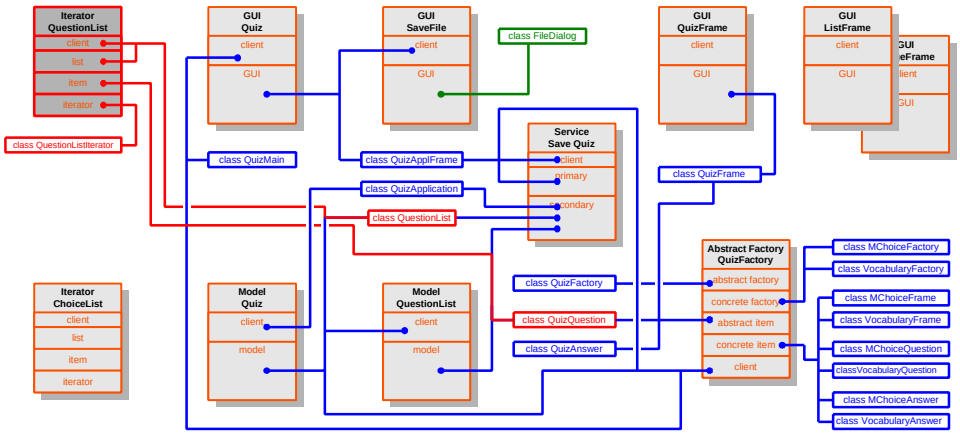
Component Insertion

SOFTWARE ENGINEERING



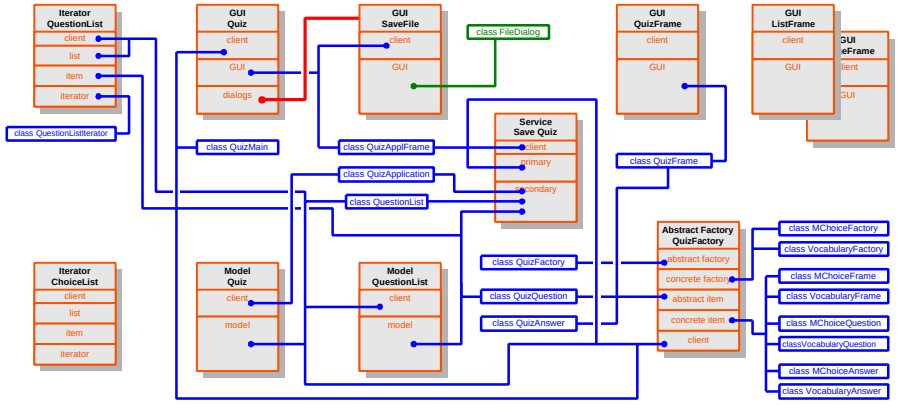
Component Deletion

SOFTWARE ENGINEERING



Relations between Components

SOFTWARE ENGINEERING



Role Constraints

SOFTWARE ENGINEERING

- Objects may
  - » play several roles at once
- Roles may
  - » require another role from their object (=object that play them)
  - » require from their object not to play another role
  - » mutually require each other
- Role constraints are specified within the context of a role model
  - » role-implied
  - » role-equivalent
  - » role-prohibited
  - » role-dontcare

Design Component Constraints

SOFTWARE ENGINEERING

- roles must be played by the same actor ⊙
- roles may be played by the same actor ◎
- roles must not be played by the same actor ⊠
- role 1 must be played by subclass of role 2 ↓
- role 1 must be played by superclass of role 2 ↑

- intra-design constraints
- inter-design constraints

Visitor Component	AV	CV	AE	CE	OS	CL
# instances	1	>1	1	>1	>0	>0
Same instantiation						
AV: abstract visitor	–	↑	⊠	⊠	⊠	⊠
CV: concrete visitor	↓	–	⊠	⊠	⊠	⊠
AE: abstract element	⊠	⊠	–	↑	⊠	⊠
CE: concrete element	⊠	⊠	↓	–	◎	⊠
OS: object structure	⊠	⊠	⊠	◎	–	◎
CL: client	⊠	⊠	⊠	⊠	◎	–
Different visitor instantiation						
AV: abstract visitor	⊠	⊠	⊠	⊠	⊠	⊠
CV: concrete visitor	⊠	⊠	⊠	⊠	⊠	⊠
AE: abstract element	⊠	⊠	⊠	⊠	⊠	⊠
CE: concrete element	⊠	⊠	⊠	⊠	⊠	⊠
OS: object structure	⊠	⊠	⊠	⊠	⊠	⊠
CL: client	⊠	⊠	⊠	⊠	⊠	◎
Other component instantiations						
Abstract factory						
Abstract factory	⊠	⊠	⊠	⊠	⊠	⊠
...						

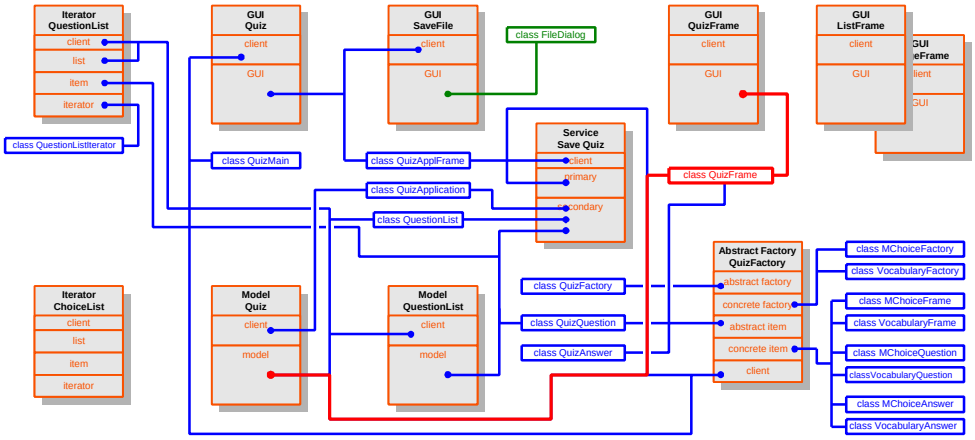
Intra-design Role Constraints

SOFTWARE ENGINEERING

```
rolemodel {
    abstract=Visitor Component
    name=default
    description: A client that uses the Visitor pattern must create...
    roletype: ...
    constraint 1, AbstractVisitor
    constraint 2+, ConcreteVisitor
    constraint 1, AbstractElement
    constraint 2+, ConcreteElement
    constraint 1+, ObjectStructure
    constraint role-prohibited, AbstractVisitor, AbstractElement
    constraint role-prohibited, AbstractVisitor, ConcreteElement
    constraint role-subclass, AbstractVisitor, ConcreteVisitor
    constraint role-subclass, AbstractElement, ConcreteElement
    ...
}
```

Simple Constraint Violation Example

SOFTWARE ENGINEERING



Role Constraints — Abstract Factory and Visitor

SOFTWARE ENGINEERING

– Abstract Factory

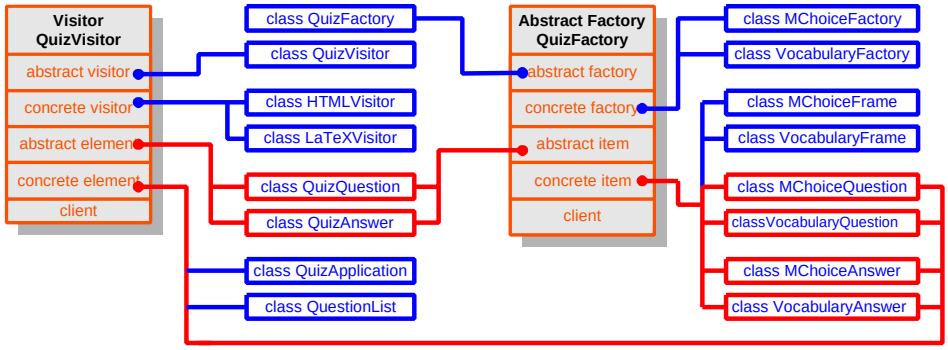
- » It isolates concrete classes.
- » It makes exchanging product families easy.
- » It promotes consistency among products.
- » Supporting new kinds of products is difficult.

Combination of roles of a visitor component and roles of an abstract factory component may indicate poor design!

– Visitor

- » Visitor makes adding new operations easy.
- » A visitor gathers related operations and separates unrelated ones.
- » Adding new ConcreteElement classes is hard.
- » Visiting across class hierarchies.
- » Accumulating state.
- » Breaking encapsulation.

Abstract Factory / Visitor Violation



Inter-design Role Constraints

```
constraint {
   rolemodels=Visitor, AbstractFactory
   description: A combination of roles of a visitor component and an abstract
         factory component can indicate poor design, because the consequences
         of these pattern's applications do not work together well. ...
   constraint role-prohibited,
         Visitor.AbstractVisitor, AbstractFactory.AbstractFactory
   constraint role-prohibited,
         Visitor.AbstractElement, AbstractFactory.AbstractProduct
   constraint role-prohibited,
         Visitor.ConcreteElement, AbstractFactory.ConcreteProduct
   ...
}
```

Role Constraints on Multiple Instantiations




```
rolemodel {
   abstract=Visitor Component
   name=default
   ...
   instances {
         constraint role-prohibited, AbstractVisitor, AbstractVisitor
         constraint role-prohibited, AbstractVisitor, ConcreteVisitor
         constraint role-prohibited, AbstractVisitor, AbstractElement
         constraint role-prohibited, AbstractVisitor, ConcreteElement
         ...
   }
}
```


Design Constraints



- constraint is a restriction on values of a model or system
- object constraint language (OCL)
  - » part of the unified modeling language (UML)
  - » used for expressing constraints on object-oriented models
- restriction of arbitrary role assignments
  - » better documentation, improved precision, better communication
  - » prohibit design flaws more effectively than the documentation of design patterns
  - » reify additional design knowledge
  - » inter- and intra-design constraints

Tool Support



- design level
  - » all information at design level
  - » development tools needed
  - » source code generated
  - » compiler integration necessary
- source code level
  - » special comments, e.g., similar to javadoc
  - » regular tools useful
  - » additional tools desirable
  - » inconsistencies possible

Documentation



- implicit documentation
  - » architecture, design
  - » role distribution
  - » design pattern documentation
  - » GUI, model, services
  - » design reuse
  - » documentation reuse
- explicit documentation
  - » design decisions
  - » design alternatives
  - » pattern deviations
  - » constraints violations

javadoc



- tool from Sun Microsystems
- javadoc input
  - » declarations and documentation comments in Java source code
- javadoc output
  - » HTML documentation describing the packages, classes, interfaces, methods, etc. of a software system
- customization with doclets
  - » specify the content and format of the output to be generated
- special tasks not related to producing systems documentation:
  - » e.g., a diagnostic doclet to check that all class members have documentation comments

Javadoc Example




```
/**
 * A class representing a simple drawing window on the
 * screen. Example usage:
 * <pre>
 *    SimpleDraw s = new SimpleDraw();
 *    s.setVisible(true);
 * </pre>
 *
 * @author      Johannes Sametinger
 * @version     1.2
 * @see         java.awt.Frame
 */
class SimpleDraw extends Frame {
   ...
}
```

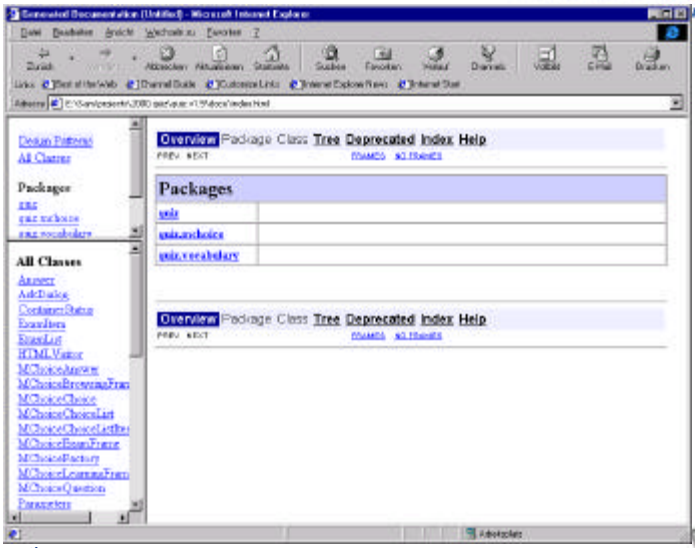
@pattern

<pattern name>.<instance name>.<role name> <text>

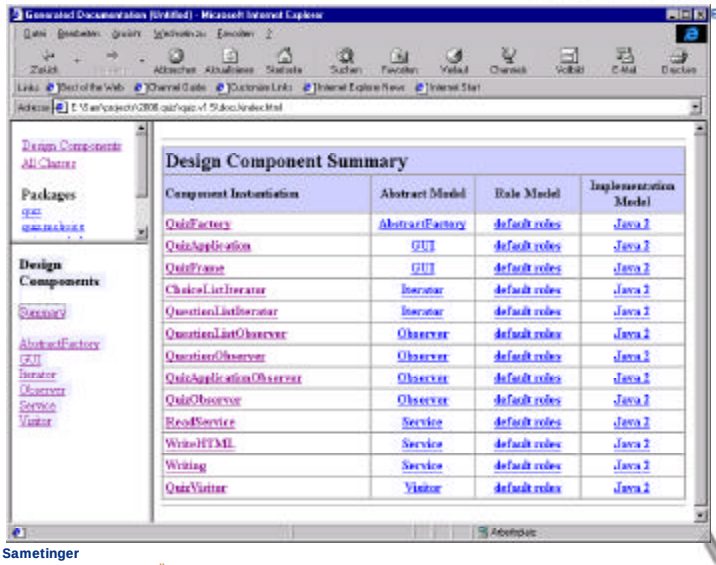
SOFTWARE ENGINEERING

```
package quiz;
/** represents the basic structure of quiz questions
 *  @author Johannes Sametinger
 *  @pattern Observer.QuestionObserver.Subject.
 *  @pattern Iterator.QuestionListIterator.Item
 */
abstract public class Question extends java.util.Observable {
    ...
/** defines an accept operation that takes a visitor as an argument.
 *  @param v Visitor to be accepted
 *  @pattern Visitor.QuizVisitor.ConcreteElement.
 *  this is the abstract method for all subclasses
 */
   abstract public void accept(Visitor v);
}
```

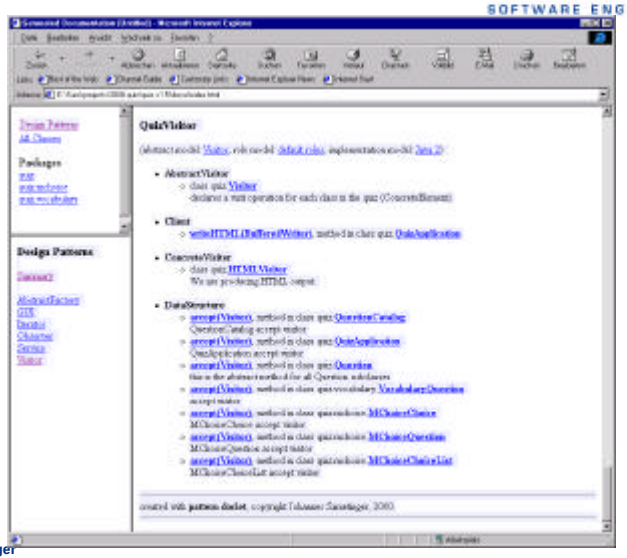

Javadoc Output



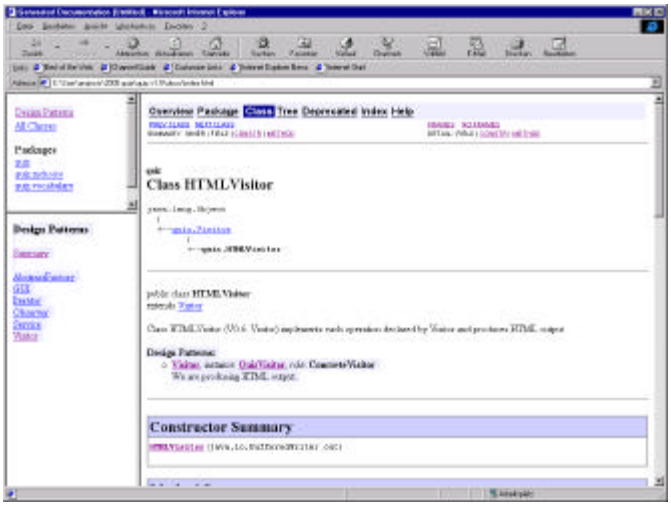
Pattern-Specific Output



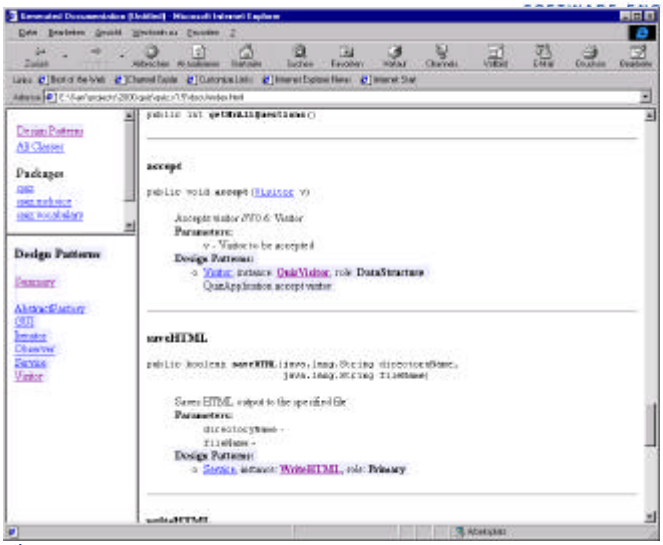
Design Documentation



Class Documentation



Method Documentation



Related Work — Aspect-Oriented Programming

SOFTWARE ENGINEERING

- Aspects
  - » capture design issues that cannot be adequately expressed otherwise (code scattered throughout the system)
  - » programming languages do not provide abstraction and composition mechanisms for several design issues, i.e., for all kinds of units a design process breaks a software system into
  - » cover only specific design aspects, but can be generic in that they can be applied to classes and methods with certain properties.

  [Kiczales et al. 97]
- Design Components
  - » no support of genericity (for now)
  - » We may have to include this ability in the future, but currently we think that capturing static aspects is sufficient for major design issues.

Related Work — Architecture Description Languages

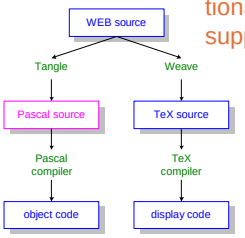
SOFTWARE ENGINEERING

- **Architecture Description Languages**
  - » provide notations for the description of software system structures
  - » model components, connectors, roles, ports, bindings, configu-rations
  - » model in terms of hierarchical configurations and interacting components
  - » examples: Darwin, UniCon, Aesop, Wright
- **Design Components**
  - » model architectural knowledge to some degree
  - » no experience yet
  - » will be less powerful than existing architecture description languages
  - » compositional reusability may be an advantage

Related Work — Literate Programming



- Donald E. Knuth, 1984
  - » structured programming + structured documentation
  - » "Do not try to instruct the computer what to do, but rather try to tell humans what you want the computer to do."
  - » aimed at system documentation
  - » leads to more carefully constructed programs
  - » leads to better and more relevant system documentation



- Design Components
  - » Literate programming sections correspond to service component instantiations.
  - » Literate programs can explicitly describe design issues like patterns.
  - » There is neither a compositional support nor is there any support of constraints.

Related Work — Framework Role Modeling



- **Framework design approach with role modeling**
  - » explicit description mechanisms for role models, role types, role constraints, frameworks, layers, class models, extension points, free role types, built-on classes
  - » extensive means for design descriptions, especially for frameworks

[Riehle 2000]

- **Design Components**
  - » primarily concentrate on compositional reuse
  - » model only reusable design aspects
  - » leave out framework and layer issues
    - – frameworks and layers are too specific and extensive to have their design reused as a whole, i.e., for the development of other frameworks or layers

Related Work — Library Design Patterns



— **Library design patterns**
- » easy access of design patterns in a library
- » apply design patterns by inheriting from classes in library
- » hard to adapt in other ways than those foreseen
- » fixed use of names

[Agerbo, Cornils 98]

— **Design Components**
- » flexible adaptation
- » no constraints on names
- » no need for (multiple) inheritance
- » better documentation

Related Work — Layered Modeling



— **Layered modeling of design patterns**
- » **Role models** express patterns in terms of abstract state and behavioral semantics, thus, capturing the spirit of a pattern without non-essential details.
- » **Type models** add domain-specific refinements.
- » **Class models** represent deployment of the type model in application-specific terms.
- » Constraint information in terms of sets, upon which constraints applying to set members are specified

[Lauder, Kent 98]

— **Design Components**
- » also use role models to describe the structure of cooperating objects
- » role and type model ~ role model, class model ~ concrete model
- » no explicit refinement of role models into type models
- » additional instantiation model with application-specific information
- » less formal approach for constraints

Conclusion

SOFTWARE ENGINEERING

- design expertise
- design constraints
- design blurring
- documentation

- What we already have:
  - » concepts available
  - » first evaluation/case study

- What we still need:
  - » basic set of design components
  - » infrastructure, tools
  - » further case studies
  - » properties of good/bad designs
  - » design constraints
  - » concrete models by platform vendors
  - » design properties of component-based systems

www.swe.uni-linz.ac.at/sametinger