Applications on the basis of a Framework for the Finite Element Method (FEM)

Bauhaus-Universität Weimar, Informatik im Bauwesen

2009

Contents

Applications on the basis of the Framework for the Finite Element Method (FEM)

1 Installation

Application software on the basis of the FEM-Framework may be downloaded as a zip-file that is provided under the name ***FEM.zip***. Root directory for the installation is *FEM*. The zip-files contains:

- the source files (subdirectory *source*) for various FEM-applications which can either be compiled separately in an individual environment or imported into an IDE (e.g. Eclipse),
- the class files (subdirectory *binary*) for applications to be run directly without recompilation,
- some input files (subdirectory *input*) for example calculations to be performed,
- the Java-documentation (subdirectory *doc*) and
- some class notes (subdirectory *notes*) for explanation of the fundamental equations and solution strategies.

Currently, three different applications are provided

- linear elasticity analysis (subdirectory *source/elasticity*),
- linear stationary and transient heat transfer analysis (subdirectory *source/heat*) and
- linear static and dynamic analysis of structural beams and frames (subdirectory *source/structuralAnalysis*).

Finally, a 2D-CAD application is provided (subdirectory *cad*) for integration of analysis into an interactive CAD-environment. The CAD-software used for these purposes is openly available under the name *cademia* (http://www.cademia.org).

If not only access to the source files is required but if major development efforts are to be undertaken, an "Integrated Development Environment – IDE" is a preferable choice. A powerful and free IDE is available, for example, under "Eclipse IDE for Java Developers" (http://www.eclipse.org/downloads/). In this case, the zip-file with the source files may be unpacked and the sources may be imported into the IDE.

The additional overhead of installing and learning an IDE may seem prohibitive to many users but over time with many enhancements and modifications to be performed, the additional investment quickly pays off. New efforts often start by "only needing to do little development" but quickly turning into something much more complex as initially intended.

It is our experience that even students in class quickly appreciate the additional help and support from an IDE over the tedious development using console input with a context sensitive editor and a standard compiler invocation. They are ready to accept the additional efforts for learning an IDE and this even motivates them better.

2 Formulation of the FEM Solution

The analysis of problems in any application consists of the Finite Element approximation of the system equations with corresponding formulations of the element equations, the external influences on the model (loads), the constitutive relations (material properties) and the external boundary conditions (support).

Specific element definitions with corresponding material, load and support definitions must be chosen that are appropriate for the problem to be solved defining the geometry of the problem (element geometry) and the element characteristics for the model.

The Framework provides several definitions for element geometries that are totally independent of specific applications. These include one-dimensional elements with two nodes in 1d-space, one-dimensional elements with two nodes in 2d-space, linear triangular elements with three nodes in 2d-space, linear quadrilateral elements with four nodes in 2d-space and linear volume elements with eight nodes in 3d-space. These may serve as a basis for deriving individual, own applications for any user.

Model input and output must also be defined in an appropriate way. Input is most commonly achieved via model definition in an input file and output may be defined in a simple alphanumeric form or in a more intuitive graphical form.

3 Implementation on the basis of the FEM Framework

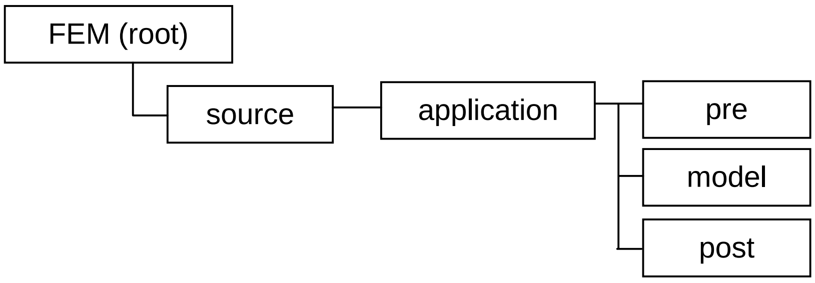
Applications of the FEM for the solution of physical problems are embedded within the context of the general FEM Framework defined before.

In this context, all applications are stored in the hierarchical structure of the FEM Framework under the hierarchy *source/application*.

Any application must define

- a **main program** for controlling a specific application analysis
  → contained in the root hierarchy of the specific application under *application*,
- a **file parser** for parsing the input file containing the persistently stored model data
  → contained in the hierarchy under *application.pre*
- several classes defining the complete **model** information including element, material, load and support information
  → contained in the hierarchy under *application.model*
- a class for **output** of information on the console and optionally on a graphical screen
  → contained in the hierarchy under *application.post*

Any new application is embedded into the hierarchical structure of the FEM Framework in the following way:

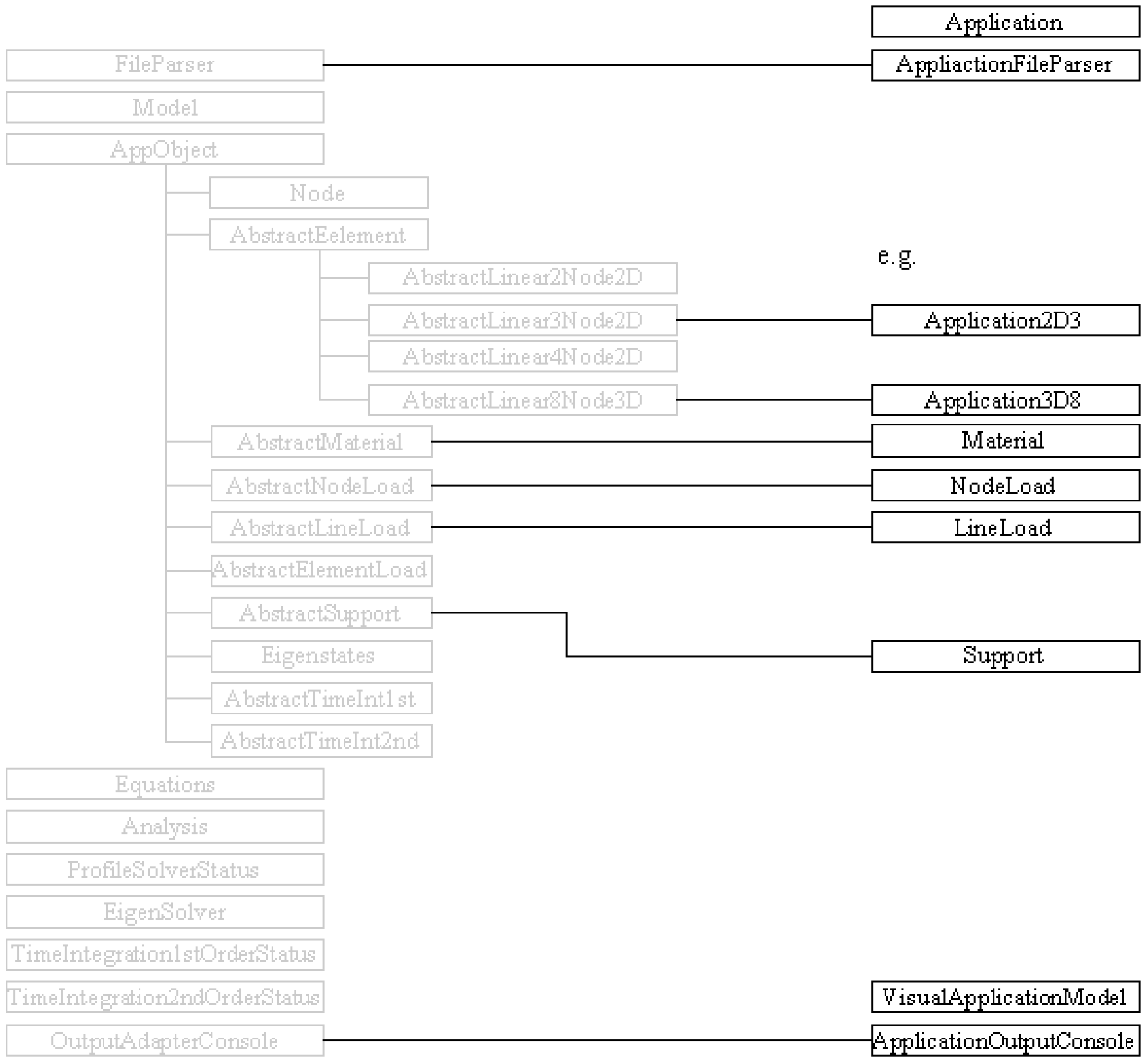


The implementation of classes defining a new application using FEM consists of a file parser for reading a new model, of the application specific element, material, load and support definitions, optionally of the definitions for time integration of the model and of the respective functionality for output of system results.

It is important to note that the mathematical description of the physical problem is almost exclusively restricted to the classes in the model.

The new classes are embedded into the class hierarchy of the FEM Framework in the following way:

Class Definitions (Application) on the basis of the Framework:



3.1 Main programs for application analysis:

Main programs for the analysis of specific applications are implemented in class **Application**.

Applications need to import general Java functionality of File input and output (File IO). They further need to import functionality of the FEM Framework for the model and the analysis and functionality for file handling contained in the FEM utilities. Finally, functionality for parsing input information and displaying model definition and behavior for a specific application need to be imported.

```
package heat;

import java.io.File;

import framework.model.Model;
import framework.model.Analysis;
import util.FileHandling;

import heat.post.ApplicationOutputConsole;
import heat.pre.ApplicationFileParser;
```

Class Application defines objects for the model, analysis and output as static attributes.

Method main starts by invoking method *getInputFile* from utility class *FileHandling*. The selection of applicable input files is accomplished via a file-chooser dialog box. Method *getInputFile* will require two arguments, one for specifying storage location of the input file relative to the root-directory, i.e. FEM, and a header definition for the corresponding dialog box.

The essential part of method main is defined by the flow of the analysis enclosed in a try and catch-block for possible exceptions generated during analysis.

First, a new model object is generated on the basis of input data as a result from parsing the input file. A message will report the successful execution of file parsing.

Next, a new object *output* of class *ApplicationOutputConsole* is generated for the specific model to be analyzed. Method *output* of this object will allow for checking model input.

Next, a new analysis object is generated for a specific model. The analysis object will provide functionality for a specific type of analysis, e.g. method *systemMatrix* for assembling the system matrix for the model, method *computeSystemVector* for generating the system vector and method *solveEquations* for solving the system equations. Results of the analysis are stored in object *model*. Successful execution is acknowledged by a corresponding message.

Next, object *output* of class *ApplicationOutputConsole* is used for output of results of the model analyzed.

```
public class Application  {
    private static Model model;
    private static Analysis analysis;
    private static ApplicationOutputConsole output;

    public static void main(String args[ ]) {
    File file = FileHandling.getInputFile("./input/application", "FEM input files", "inp");
    try {
         // setup and visualize model, analyze model and output results
         model = new ApplicationFileParser(file).getModel();
         output = new ApplicationOutputConsole(model);
         output.output();

         analysis = new Analysis(model);
         analysis.systemMatrix();
         analysis.computeSystemVector();
         analysis.solveEquations();

         output.output();
         System.exit(0);

    } catch (Exception e) {
         e.printStackTrace();
         System.err.println(e);
} } }
```

3.2 Parsing persistent model information (input file)

Class ***ApplicationFileParser*** must be provided for parsing of model information that is persistently stored in an input file in Unicode format. Generally, very powerful functionality exists for file parsing in Java based on a particular grammar. For the purposes of this implementation it was decided not to use such a general functionality but rather to use an individual implementation as the number of keywords needed is very limited and the structure of the input file very simple.

The implementation is stored in *package application.pre*. It needs to import general Java functionality about Java file input and output (IO). It further needs to import functionality of the FEM Framework with respect to *Model* objects, the *FileParser* and corresponding exceptional conditions. It finally needs to import application specific functionality for FEM components as there are elements, influences, materials and supports. Separate sections of the input file are defined for each individual set of components. Each section is defined by a preceding keyword.

Class *ApplicationFileParser* is derived from class *FileParser* of the Framework and can thus utilize **keywords** defined in that class with the corresponding **number of arguments**:

```
// public void parseIdentifier() throws IOException, ParseException
//      identifier                          1 modelName

// public void parseDimensions() throws IOException, ParseException
//      dimensions                          2 spatialDimensions, nodalDegreesOfFreedom

// public void parseNodes() throws IOException, ParseException
//      nodes                               1: number of nodal degrees of freedom
//                                          2: name, x
//                                          3 name, x, y
//                                          4 name, x, y, z
//      uniformNodeArray   dimension=1      4 nodeInitial, x, xInterval, nIntervals
//                         dimension=2      7 nodeInitial, x, xInterval, nIntervalsX,
//                                            y, yInterval, nIntervalsY
//                         dimension=3     10 nodeInitial, x, xInterval, nIntervalsX,
//                                            y, yInterval, nIntervalsY,
//                                            z, zInterval, nIntervalsZ
//      variableNodeArray  dimension=1      ? meshSpacings
//                                          2 nodeInitial, x
//                         dimension=2      ? meshSpacings
//                                          3 nodeInitial, x, y
//                         dimension=3      ? meshSpacings
//                                          4: nodeInitial, x, y, z

// public void parseEigenstates() throws IOException, ParseException
//      eigenstates                         2 name, numberOfStates
```

Additional keywords required in class *ApplicationFileParser* must be defined as needed.

The process of parsing input for a FEM analysis from a file is rather schematic:

- A **keyword** identifies a set of corresponding data (lines) to be read,
- each subsequent line contains a specified **number of arguments** defining the information needed for generating a new FEM component. Each line is analyzed item by item (token by token),arguments are separated by predefined separators, i.e. blank, tab, comma
  StringTokenizer tokenizer = new StringTokenizer(s, " \t,");
- the constructor of the new component is invoked with the items as parameters and
- this is carried on until an empty line is encountered completing the particular keyword section.

Once a keyword section is completed, method *reset* will cause reading the input file from the beginning again for the next keyword to be processed.

Keyword **identifier** is defined in the framework. It expects the mandatory input of a string for a new model identifier, otherwise an exception will be thrown. The string will be passed to the constructor of a new object of class model.

Keyword **dimensions** is defined in the framework. It expects input of two integer numbers indicating the spatial dimensionality of the problem to be solved and the number of nodal degrees of freedom (e.g. 2   1 defines a 2-dimensional problem with 1 degree of freedom per node).

Keyword **nodes** is also defined in the framework. It expects either an integer number for the applicable number of nodal degrees of freedom for all subsequent nodes or a string for identifying a new node object followed by float numbers for the definition of the nodal coordinate values. This keyword is generally followed by a number of lines defining the complete geometry (nodes) for the model.

Nodes may also be generated. For this purpose two different keyword are available.

uniformNodeArray will generate a mesh of equally spaced node components. This is accomplished by entering an initial identifier for all nodes to be generated, followed by starting coordinate(s), the spacing(s) of the mesh and the number of intervals in each direction. This process is supported for 1, 2 and 3 dimensions in which case the starting coordinate, the mesh spacing and the number of intervals need to be specified for each dimension. Unique identifiers for each node will be generated based upon the initial identifier which will then be concatenated with integer numbers for each successive interval, e.g.

uniformNodeArray			
N	0.	2.	10

will generate an array of 11 nodes at 10 spacings of 2. units from the origin (0.).

New nodes will be generated with respect to the origin specified (e.g. 0.) Each node will have a unique identifier based upon an initial identifier (e.g. N).

variableNodeArray will be generated based upon a sequence of mesh distances to be entered in the first line after the keyword *variableNodeArray*. The mesh distances to be specified have to start with 0 and sequence of distances with respect to the origin (initial coordinates). The number of distances entered will determine the number of nodes to be generated. For this purpose, simply an initial nodal identifier and nodal coordinates for the mesh origin need to be entered.

example:

variableNodeArray				
0.	1.	3.	7.	15.
N	0.	0.	0.	

Will generate an array of nodes at mesh spacings of 0, 1, 3, 7 and 15 units from the origin for each spatial dimension.

New nodes will be generated with respect to the origin specified (e.g. 0., 0., 0.) Each node will have a unique identifier based upon an initial identifier (e.g. N).

In case of **time integration problems** additional keywords are needed for entering the number of eigenstates to be considered, time integration parameters, initial temperatures and an optional forcing function.

Keyword **eigenstates** is defined in the framework. It expects an integer number for the number of eigenstates to be considered.

Separate methods are defined for processing input information. Each starts by returning to the top of the file and keeps reading new lines until the respective keyword is encountered. Once a particular keyword is found, all subsequent lines are read in sequential order and each line is checked if the number of arguments is according to definitions, otherwise a corresponding exception is thrown. Parsing a keyword section is completed when an empty line is encountered.

Method *parseElements* is defined for parsing application elements of the FEM model. It throws an exception if a given element identifier already exists and if the node identifiers or the material identifier that an element refers to are undefined.

Method *parseMaterials* is defined for parsing materials with the constitutive relations needed for the FEM model. It throws an exception if a given material identifier already exists.

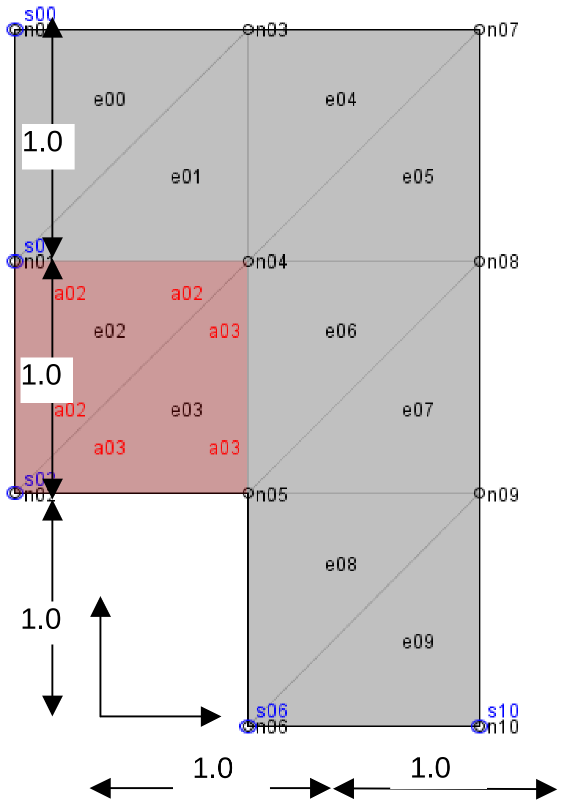
Method *parseLoads* is defined for parsing external influences (loads) on a model. It throws an exception if a given load identifier already exists.

Method *parseSupports* is defined for parsing support conditions of an FEM model. It throws an exception if a given support identifier already exists.

This is demonstrated for an example of linear heat transfer with triangular elements.

e.g.

```
identifier
wall corner with 9 elements

nodes
1
n00    0.0    3.0
n01    0.0    2.0
n02    0.0    1.0
n03    1.0    3.0
n04    1.0    2.0
n05    1.0    1.0
n06    1.0    0.0
n07    2.0    3.0
n08    2.0    2.0
n09    2.0    1.0
n10    2.0    0.0

heat2D3
e00    n00    n01    n03    iso
e01    n01    n04    n03    iso
e02    n01    n02    n04    iso
e03    n02    n05    n04    iso
e04    n03    n04    n07    iso
e05    n04    n08    n07    iso
e06    n04    n05    n08    iso
e07    n05    n09    n08    iso
e08    n05    n06    n09    iso
e09    n06    n10    n09    iso

materials
iso    5.0

areaLoad3
a02    e02    30.0
a03    e03    30.0

supports
s00    n00    10.0
s01    n01    10.0
s02    n02    10.0
s06    n06    20.0
s10    n10    20.0
```



In the example above, **nodes** with a single degree of freedom are defined with the external identifiers *n00* to *n10* and corresponding 2-dimensional coordinates. Elements **heat2D3** are defined with the external identifiers *e00* to *e09* and material *iso*. **materials** are defined with the identifier *iso* and a double value for the specific property, e.g conductivity. **areaLoad3** are defined with the identifier *a02* and *a03* and finally **supports** are defined with the identifier *s00, s01, s02, s06, s10* and a double value for the specific support value (e.g. predefined temperature).

Relations to other objects are defined by referring to the corresponding external string identifiers. For the example, element *e00* is composed of nodes *n00*, *n01* and *n03* and has material *iso*. Elements *e01* through *e09* are defined accordingly. Area load *a02* is associated with element *e02* and support *s00* is associated with node *n00*.

Using external identifiers for establishing object relations has the considerable advantage that these relations may be established without the necessity that all objects referred to must have been generated before. Initially, all relations are established solely on the basis of the external string identifiers without any knowledge about internal storage allocation (internal references). Later, when starting the application, a link between external identifiers and internal references must be generated and used for referring to the actual internal addresses of the objects.

In the context of the FEM Framework this approach allows to strictly separate between topology and geometry. The topological model (elements referring to nodes that are only defined by name yet, loads, supports) may be defined completely separate from the geometrical model (nodes with coordinates).

The persistent data structure is stored in an Unicode-file as outlined above. A file parser is provided for interpreting the contents of the persistent data structure and for generating the corresponding transient structure. The transient data structure is based upon Java container structures.

3.3 Model information for the analysis

Model information for the analysis of linear heat problems consists of the Finite Element approximation defining the element equations, the influences on the model (loads), the constitutive relations (material properties) and the external boundary conditions (supports).

Furthermore, a specific element must be chosen that is appropriate for the problem to be solved defining the geometry of the problem (element geometry) and the element characteristics for the model.

Each application that is defined in the context of the FEM Framework simply needs to implement the additional application specific functionality for

- the element matrices,
- the material properties,
- the external influences on the model,
- the support conditions for the model and
- the time integration information.

Any information in the context of a FEM model based upon the FEM-Framework must be recognized as general application objects. These must conform to some general rules that are essential for use in the Framework.

General Application Objects

These classes will be used by the parser in order to generate application specific objects.

Any application specific object defined on the basis of the Framework must fulfill ***interface IAppObject***.

```
public interface IAppObject {
    public String getId();
}
```

Any application object that may be referenced by other objects must fulfill ***interface IReferencedObject***.

```
public interface IReferencedObject extends IAppObject {
        public void setReferences(Model model) throws FemException;
}
```

This is often inherently accomplished by having individual object definitions implementing predefined object interface definitions, i.e. *IElement*, *IMaterial*, *ILoad*, *ISupport*.

This process is further alleviated by deriving individual object definitions from predefined abstract definitions, i.e. *AbstractElement*, *AbstractMaterial*, *AbstractElementLoad*, *AbstractNodeLoad*, *AbstractLineLoad*, *AbstractSupport* which are all defined to implement the required interface definitions.

Further specification may be provided by deriving elements from predefined element definitions like *AbstractLinear3Node2D* which will be derived from *AbstractElement*.

Any application object on the basis of the Framework must be derived from ***class AppObject***. *AppObject* basically only manages unique identification of objects based on a string Id and implements interface Comparable for ordering application objects.

```
// -------------------------------------------------------------------------
// CLASS : AppObject - Super class for all components of the FEM model,
//                               that are stored in the central objectMap
// -------------------------------------------------------------------------
// public class AppObject implements IAppObject, Comparable<Object>, Serializable
//
// protected AppObject(String m_id)
// public String getId()
// public int compareTo(Object object)
```

3.3.1 establishing element matrices

Any element of a specific application must inherently implement ***Interface IElement***, which in general defines a number of methods for retrieving node references and system indices of node objects. The final method defines the computation of the element matrix and the diagonal matrix for time integration analysis. If the element is derived from element definitions of the Framework, this reduces to implementation of *computeMatrix* and *computeDiagonalMatrix*.

```
public interface IElement extends IReferencedObject {
    public INode[ ] getNodes();
    public int[ ] getSystemIndices();
    public double[ ][ ] computeMatrix() throws FemException, AlgebraicException;
    public double[ ][ ] computeDiagonalMatrix()
                      throws FemException, AlgebraicException;
}
```

Element definitions are generally derived from ***class AbstractElement*** providing implementations of the following methods:

```
// -------------------------------------------------------------------------
//  CLASS : Abstract Element
// -------------------------------------------------------------------------
// public abstract class AbstractElement extends AppObject implements IElement
//
// public AbstractElement(String id, String[ ] eNodes, String m_materialId, int nDOF)
// public AbstractElement(String id, String[ ] eNodes, String m_crossSectionId,
//                                  String m_materialId, int nDOF)
// public String getElementId()
// public String[ ] getNodeId()
// public INode[ ] getNodes()
// public String getMaterialId()
// public int getElementDOF()
// public int getNodesPerElement()
// public int[ ] getSystemIndices()
// public void setNodeId(String[] nodeId)
// public void setMaterialId(String value)
// public void setCrossSectionId(String value)
// public void setReferences(Model m_model) throws FemException
```

3.3.2 defining material properties

The constitutive relations (material description) in FEM applications is given in general by the **material matrix E**. The material matrix is then needed for the evaluation of the element matrices.

Class Material is contained in package *application.model* and imports class *model.implementation.AbstractMaterial*. It is derived from class *AbstractMaterial*.

It simply consists of a constructor for passing the material identifier to the model and for storing the material property under a specific material name, e.g. “Conductivity”, and it provides a method for retrieving the material property (e.g. *getConductivity*).

3.3.3 defining external influences on the system

External influences on the model can either be defined as

- nodal influences
  are concentrated influences at the nodes of the element mesh. The intensity of the influence is given.
- line influences
  are line loads between two neighboring nodes of the element mesh. The line heat intensity is defined. If the line load intensity is constant, the value is given. If it varies linearly along the line, the intensities at the end nodes are given.
- area influences
  are surface influence sources with a given intensity. If the influence intensity over an element is constant, the constant value is given. If the influence intensity varies linearly across the element, its value is given at each of the nodes of the element.

Any element of a specific application must inherently implement ***Interface ILoad***, which in general defines a method for computing the element load vector and another one for retrieving the corresponding system indices. If the load object is derived from load definitions of the Framework, this reduces to implementation of *computeLoadVector*.

```
public interface ILoad extends IReferencedObject {
    public double[ ] computeLoadVector();
    public int[ ] getSystemIndex();
}
```

Definitions for external influences are generally derived from ***class AbstractElementLoad, AbstractNodeLoad, AbstractLineLoad*** providing implementations of the following methods:

```
// ------------------------------------------------------------------------
//  CLASS : Abstract Element (Area) Load
// ------------------------------------------------------------------------
// public abstract class AbstractElementLoad extends AppObject implements ILoad
//
// public AbstractElementLoad(String id, String m_elementId)
// public void setElementId(String id)
// public void setInElementCoordinateSystem(boolean ecs)
// public void setReferences(Model m_model) throws FemException
// public String getElementId()
// public boolean isInElementCoordinateSystem()
// public IElement getElement()
// public int[ ] getSystemIndex()
// public double[ ] getIntensity()
// public String toString()


// ------------------------------------------------------------------------
// CLASS : AbstractNodeLoad
// ------------------------------------------------------------------------
// public abstract class AbstractNodeLoad extends AppObject implements ILoad
//
// public AbstractNodeLoad(String id, String m_nodeId)
// public void setReferences(Model m_model) throws FemException
// public String getNodeId() { return nodeId; }
// public INode getNode()
// public int[ ] getSystemIndex()
// public double[ ] computeLoadVector()


// ---------------------------------------------------------------
// CLASS : AbstractLineLoad
// ---------------------------------------------------------------
// public abstract class AbstractLineLoad extends AppObject implements ILoad
//
// public void setReferences(Model m_model) throws FemException
// public String getStartNodeId()
// public String getEndNodeId()
// public INode getStartNode()
// public INode getEndNode()
// public double[] getLoadVector()
// public int[ ] getSystemIndex()
// public abstract double[ ] computeLoadVector()
```

3.3.4 setting up support conditions

Support conditions for the model can be defined for each node. Nodes in any specific applications are characterized by a specific number of “nodal degrees of freedom”. Only the number of prescribed values at a node for a specific application need to be defined and the system status vector needs to be adjusted.

Class Support is contained in *package application.model*. It is generally derived from class *AbstractSupport* which needs to be imported from *package framework.model.abstractclasses*. It also needs to implement interface *model.interfaces.ISupport*.

In case class *Support* is derived from class *AbstractSupport*, methods *getNode* and *getNodeId* are already available.

```
public interface ISupport extends IReferencedObject {
    public void setReactions(double[ ] support);
    public INode getNode();
    public boolean[ ] getRestraint();
    public double[ ] getPrescribed();
    public boolean getTimeDependent();
    public double[ ] getTimeVariation();
    public String getNodeId();
}
```

Definitions for boundary conditions are generally derived from ***class AbstractSupport*** providing implementations of the following methods:

```
// --------------------------------------------------------------------------
// CLASS : Abstract Support
// --------------------------------------------------------------------------
// public class AbstractSupport extends AppObject
//
// public AbstractSupport(String id, String m_nodeId)
// public void setReferences(Model m_model) throws FemException
// public int getNodalDegreesOfFreedom()
// public INode getNode()
// public String getNodeId()
// public String toString()
```

3.3.5 defining information for time integration

Time integration information for the model can be defined for different time integration objects. Time integration objects are defined to store all the relevant information for a specific time integration problem in the model.

Class TimeInt is contained in *package application.model*. It is derived either from class *AbstractTimeInt1st or AbstractTimeInt2nd* which need to be imported from *package framework.model.abstractclasses*. AbstractTimeInt1st is defined for time integration of 1st order differential equations and *AbstractTimeInt2nd* for time integration of 2nd order differential equations.

Both define attributes for the time step, the integration parameters depending on the method used, the filed variable and field variable derivative matrices and the forcing function.

3.4 Output of system results

Output functionality as a result of the analysis of particular application is generally implemented in two different classes, one for alphanumeric console output (ApplicationOutputConsole) and another one (VisualApplicationModel) for output in a graphical frame (window Application Visualizer) created with Java Swing technology.

Class definitions for output of information are generally derived from ***abstract class OutputAdapterConsole***. This simply defines abstract methods for output of model information (*printModel*), output of external influences on the model (*printInfluences*) and output of results of the analysis for a particular model (*printBehaviour*).

Literaturverzeichnis

[1] Handouts, Project of Civil Engineering, Software-Design for Numerical Calculation of Physical Processes (stationary plane heat transfer), Beucke, Könke, Richter, Luther, Weimar 10/2005.

[2] Pahl, P. J.: Finite Element Method, Stationary Heat Flow, TU Berlin, November 2000

[3] Pahl, P. J.: Finite Element Methoden für Physikalische Aufgaben, TU Berlin, Oktober 1991

[4] Sun Microsystems, Inc., Copyright 2006: Download The Java Tutorial, http://java.sun.com/docs/books/tutorial/.

[5] http://www.cademia.org, © 2006, Bauhaus-Universität Weimar | Lombego Systems

[6] http://www.opensource.org

[7] Zienkiewicz, O. C.: The Finite Element Method, McGraw Hill, ISBN 0-07-084072-5

[8] Bathe, Klaus-Jürgen: Finite Element Procedures, Prentice Hall, ISBN 0-13-301458-4