Impact Programmers Manual

Table of Contents

Table of Contents

Introduction

Welcome to the Impact Finite Element Program.

This program was designed to be a free and SIMPLE alternative to the advanced commercial Finite Element codes available today. The guideline during the development of the program has been to keep things clear and simple in design.

Impact has been designed to be easily extendible and modular to enable programmers a way to easy add features to the program without having to enter other parts of the code. Impact has been written in Java. This choice of language may seem strange at first, but with the recent development of Java engines, speed penalty is not that significant. On the other hand, the Object Oriented features and the high portability of Java is a clear advantage for the future.

What Is Impact?

Impact is a Finite Element Code which is based on an Explicit Time stepping algorithm. These kind of codes are used to simulate dynamic phenomena such as car crashes and similar, usually involving large deformations.

There are quite few explicit codes around which might seem strange since the other cousin (implicit finite element) are quite common. The implicit codes are used to simulate static loads in structures. Something that explicit codes does not manage very well.

History & Theoretical base

The explicit code is based on the simple formula of F=M*A where F represents a force, M is the mass of a body and A is the resulting acceleration of that body.

All the code does is to calculate the acceleration for the body, use a small step in time to translate this acceleration into a little displacement of the body. This displacement is then used to calculate a responding force since the body is elastic and can be stretched (thus creating a reaction force). This force is then used to calculate an acceleration and then the process is repeated again from the beginning.

As long as the timestep is sufficiently small, the results are accurate.

Impact is written in Java

Impact is written in Java for two reasons:

1. Java is an Object Oriented language and that suits Finite Element Programming perfectly
2. Java is clean, simple and extremely portable.

On the other hand, Java might seem like a strange choice since this is a high performance number crunching type of software and Java is not known to be competitive to ex. Fortran or C++ in this area.
True, it is slower but with the new interpreters from IBM and Sun, the built in runtime compiling actually gets the speed up quite a bit so this is not such an issue after all.

What is a Finite Element Model?

Any simulation (like a car that will be crashed) is made by a finite element model. The car is chopped up into numerous small pieces called elements. The elements are connected with each other at connection points called nodes. The model is described in an indata file which is basically a list of all the nodes and their coordinates in space, followed by a list of all the elements and which nodes each element is connected to.

Each element can stretch and move according to a formula. This formula is specific to every different type of element (there can be many different types), but the important thing is that the formula accurately can predict how the element should behave. Since each element knows how to behave, the whole model will also behave correctly.

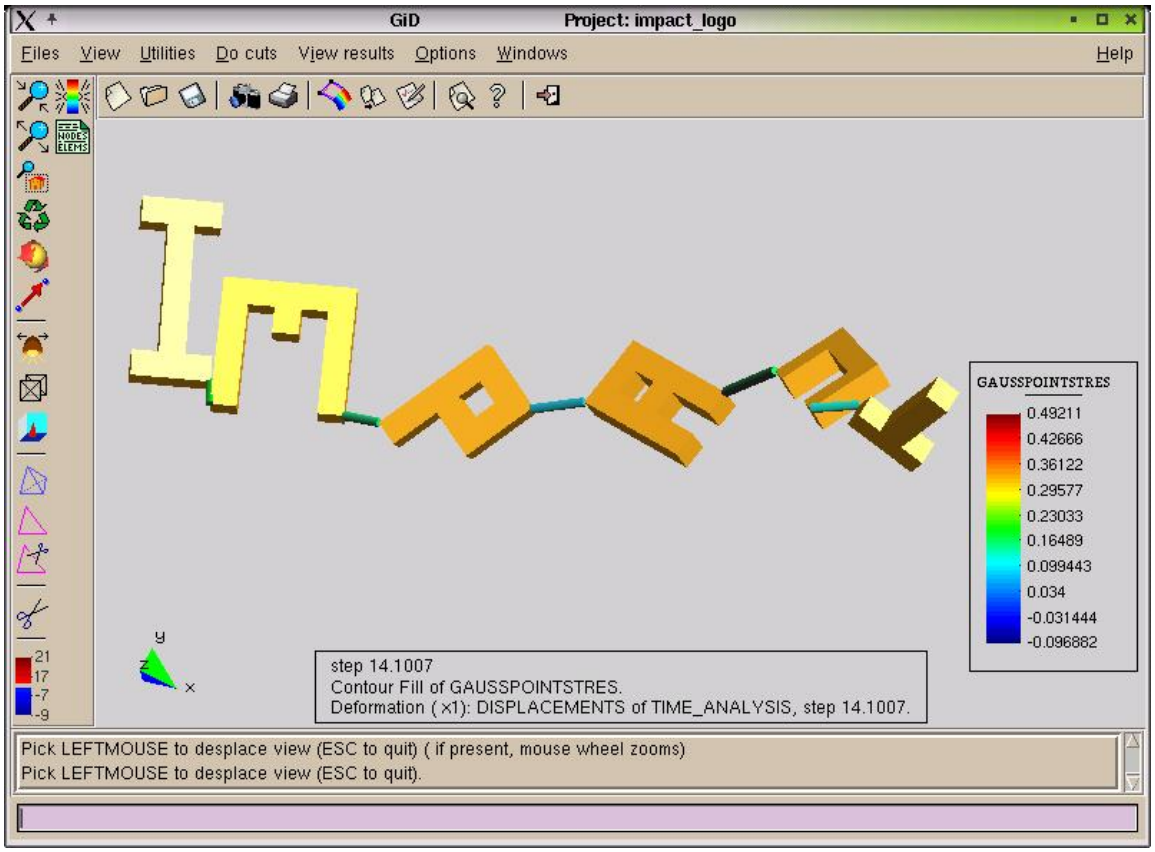
Examples of Problem That Impact Can Solve

At the moment, Impact can only handle dynamic INCOMPRESSIBLE problems. Examples of problems with this kind of limitation is basically most real world dynamic problems. The following is a list of problems that Impact will be able to solve in the future.

- Collisions of any type
- Forming operations
- Dynamic events such as chassis movement etc.

Impact is currently in alpha state. It is not validated against any other program or test results which means that results should be treated with care

Here is a picture of the Impact_Logo model included as an example problem



The model consists of rod and solid elements. A constant force is pulling the lower corner of the T while the upper corner of I is kept fixed in space.

Literature of Interest

There are a large number of books available on Finite Element Theory. Most of them describe Finite Element from a static point of view and is therefore of limited interest to the potential Impact programmer.

On the other hand, the theory of element formulation is often usable to a large extent and having that in mind, here are a few proposals:

- Concepts And Applications Of Finite Element Analysis, Third edition − Robert D. Cook, David S. Malkus, Michael E. Plesha, ISBN 0−471−84788−7
- The Finite Element Method − Linear Static and Dynamic Finite Element Analysis − Thomas J. R. Hughes, ISBN 0−484−41181−8
- Nonlinear Finite Elements for Continua and Structures − Ted Belytschko, Wing Kam Liu, Brian Moran. ISBN 0−471−98773−5

The first book is recommended to beginners and engineers in general since it deals with most issues from a linear algebra perspective. This makes the code writing quite close to the Impact format. It is also a very good book and the one I have had best feedback from.

Ted Belytschko's book is the "bible" in this field. The man behind explicit codes have finally written a compendium on the theory and some principle algorithms are also shown. However, for an engineers

perspective, this book is quite deep in its places and is more suitable as a reference than as a learning book for beginners.

There are also some papers written which are of interest:

- Explicit Algorithms For The Nonlinear Dynamics Of Shells − Ted Belytchko, Jerry I. Lin, Chen−Shyh Tsay, Computer methods in applied mechanics and engineering 42 (1984), page 225−251
- An Explicit Formulation For An Efficient Triangular Plate−Bending Element − Jean−Louis Batoz, International journal for numerical methods in engineering, Vol. 18, page 1077−1089 (1982)

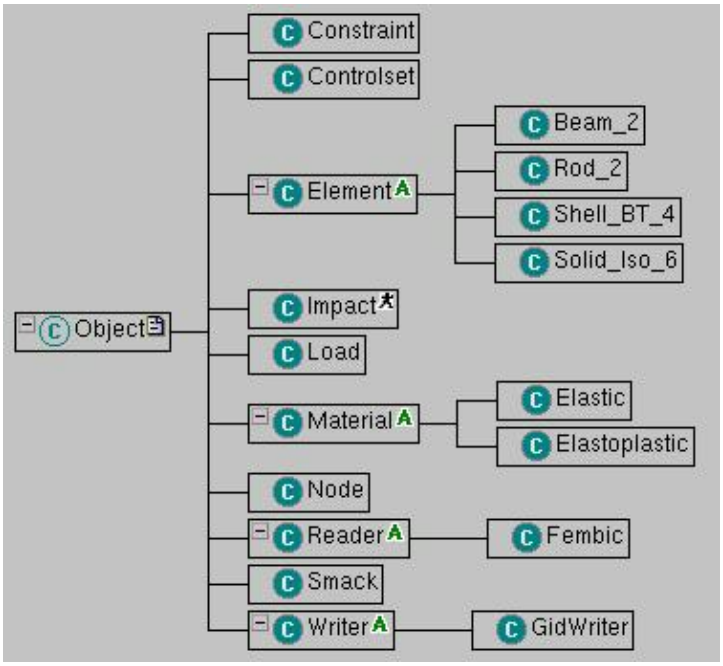
These papers form the basis of coming shell element extension to Impact.

Object oriented Java programming

To understand the concept behind object orientation, inheritance etc, the following book is a pleasure to read:

- Thinking in Java − Burce Eckel, ISBN 0−13−027363−5; The book is also available for free download at: http://www.bruceeckel.com

The Structure of Impact



The picture shows how Impact is structured. Impact is strictly object oriented in its design and I will use quite a few of the terms in the coming explanation. To appreciate the terms of object oriented programming, please read any book on the subject. Have a look in the Literature section for recommendation.

The elements and the nodes have each got their own class. Since there can be several types of elements, the element class is a superclass and it is also abstract which it acts as "format" for how a subclass should be written, which methods it should have etc. The element super class also contains some methods which are general and usable by all the element classes.

Impact is the main class. All it does is to immediately initiate an instance of the class smack and then it tells that instance to initiate, solve and postprocess the problem. Smack also includes a large array of nodes and elements which in fact is the actual model. When asked to solve the problem, it runs down these arrays and tells each element to update itself, over and over again.

Another important class is the material class. Since there can be several different types of models for a material, this is a superclass and then there are several subclasses. Right now, there are two material models available. One for pure elastic materials and one elasto−plastic materials (materials which are elastic to a point and then acts as gum beyond that point).

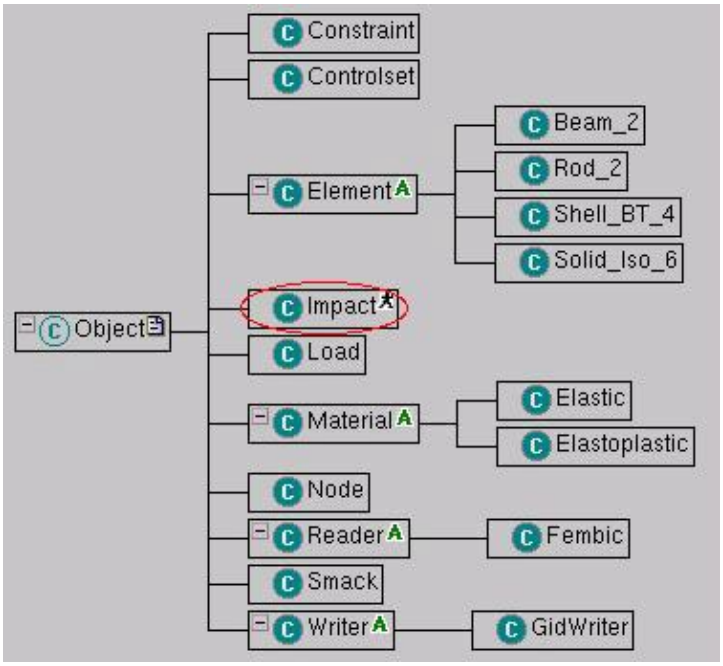
A model also contains constraints, forces and boundary conditions. Each of these have their own class and each element which has forces defined on it, knows about that by having a handle to the respective constraint etc.

Each element also knows about which nodes it has so that it can feed it's mass and forces to the nodes when asked to.

Finally, the reader class handles the reading of indata files (where the problem is defined) and the writer class handles the writing of the result files. Both are super classes in order to allow for several different indata and

outdata file formats. This allows Impact to be compatible with other programs such as LS−Dyna and Radioss in the future.

Structure of the Impact Class

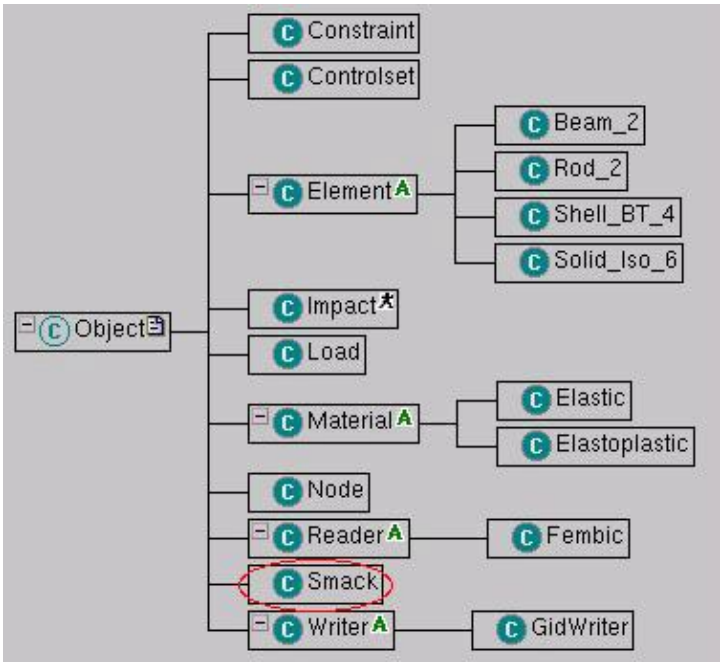


The Impact class is the main class of the program. It is here the program starts and the following happens:

1. An object from the Smack class is created
2. The object is told to initialise and read the indata file
3. It is told to solve the problem
4. It is told to finish and close.

In the future, something like a command centre can designed in this class, creating a nice user interface. The possibility of initiating several solutions running in parallel is also easily accomplished. Just create several objects from the Smack class!

The Structure of the Smack Class



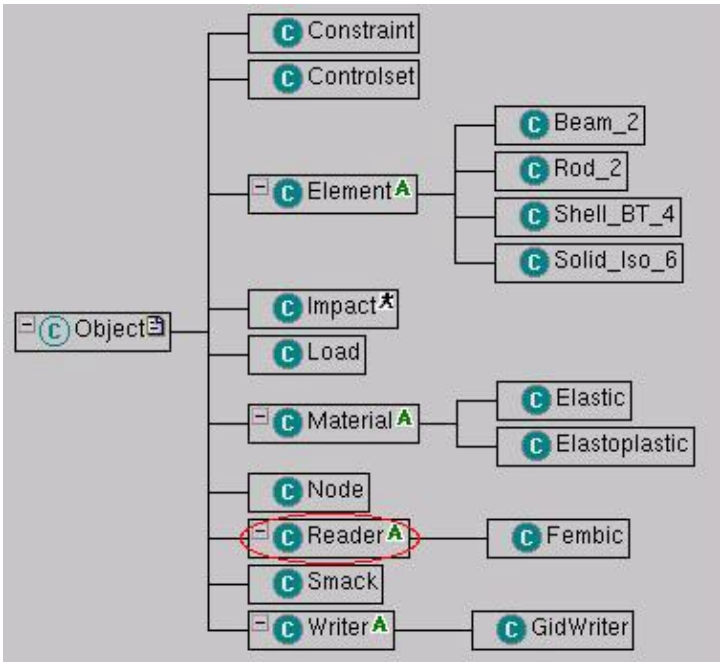
The Smack class is the centre of Impact. Here, the model is stored in two large arrays. One array for the Elements in the model and one array for the Nodes.

In addition, a number of smaller arrays are also used to store the Loads, Constraints and Materials that are defined in the problem indata file.

In short, an object created from the Smack class will do the following:

- Create a Reader object and tell it to open the indata file
- Create a Writer and tell it to open the outdata file
- One by one, ask the reader object to generate node, element, constraint, load and material objects of the right type and give them to Smack to put into its arrays.
- Tell all elements and nodes to initialize themselves
- Run through the array of elements, telling each of them to update itself by calculating its strains, stresses and nodal forces
- Run through the array of nodes, telling each of them to calculate it's new position using the forces that it has received from each element.
- Repeat the last two actions until the solution is finished, while printing every now and then.

The Structure of the Reader Class



The reader class is the abstract parent class for different file translators. The challenge with structuring Impact was to figure out a good way of separating the input file format from the rest of the program. This was the result.

The reader class (or rather any chosen subclass) have two main tasks:

- Scan the chosen input file and answer questions about how many elements it contains etc.
- Walk through the input file and generate objects (like elements or nodes) of suitable type and then return them to Impact

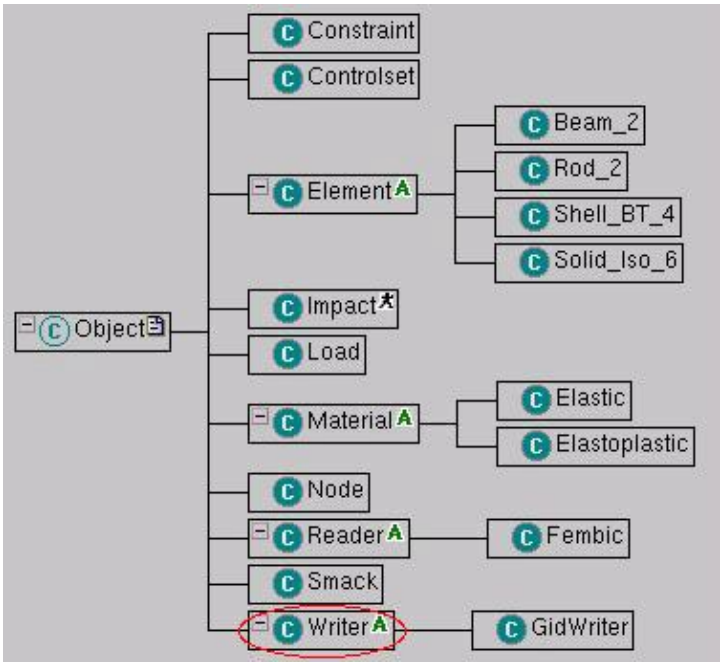
These tasks mean that the reader class must be able to interpret the indata file in terms of grammar and words.

Imagine now, that the reader class reads a typical indata file. A problem was that if the reader class must be able to know all the parameters that a certain element needs (for example), read these, generate the corresponding element object and then feed that object with all the data from the file, this means that the programmer must go in the the reader class and change when he/she adds a new element class or changes a class.

It is more attractive if the reader class can be independent from the element class. The solution used in Impact is that part of the interpreter is actually included into the element class, namely the part that reads the element parameters. The reader class just feeds the element with the actual indata file line where the parameters are written and the element does the rest.

You can read more about this solution in the element chapter

The Structure of the Writer Class



The writer class is the mother class of all the different writer classes. Each subclass provides a different output format.

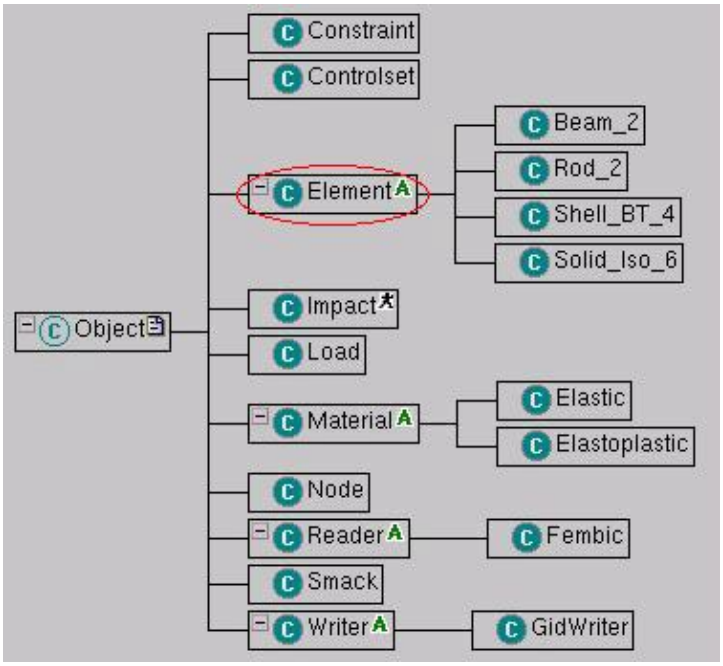
A writer object is created at the start of the solution run and it is then called every time that Impact wants to print to an outdata file. This can happen several times during a solution. The writer class then walks through the database of elements and nodes and asks each of them to write their status.

An equal challenge as for the reader class is present here regarding being able to add new element classes without having to go into the writer class and change. The solution is the same, namely that the writer class asks the element object to print it's own line of outdata (in the right format), which is then directly written to the outdata file. All the preparation is made inside the element itself.

Due to how the outdata file is structured, the method that writes the data in the element is somewhat complex. The writer calls the same method several times, depending on if it wants the element to write data for a header or for another location in the outdata file. This is accomplished using control codes and is more explained in the element chapter.

This layout means that the element class needs to have one method for each new reader class (output file format), but since the amount of file formats are significantly less than the amount of element types, this is feasible.

The structure of the Element class



The element class has been carefully designed to be self containing and easy to expand. The problem when writing a program is usually that when a programmer wants to add another feature (such as a new element type), it also means digging around in a lot of code which is indirectly related to the new element type.

The benefit of Impact is that this is not the case here. The programmer only needs to consider the following:

- Do I need a special material law or can I use any of the ones available?
- What input syntax should the new element use?

Once that is decided, the programmer is then free to add the new element along the lines described later in this manual. The basic method is to add a new subclass to the existing abstract class "Element" and then fill that new subclass with code.

When Impact is run and a number of elements are read in from the input file, a corresponding number of element objects are created. One object for each element.

The object will get a "memory" where it remembers which nodes it has attached to it, which material it is made of and so on.

If we stop a bit at the material "memory", I'd like to explain a little bit more in detail how it is handled. Some material laws need a memory in itself. This is the case since once some deformation has been made, a permanent deformation remains after unloading. This phenomenon means that the element (and material law) needs to remember how much it has been loaded in the past to accurately model the permanent deformation.

How to solve this? Well, the solution is quite simple. Once the element object has been created, a local internal material object is created which is based on the material law that the element is supposed to use. One can then think of the element object actually encapsulating it's own material object.

From here on, the element object talks only to it's own material object and since the material object keeps track on where it has been, the element object also has a memory of how much it has deformed.

Other solutions

A Solution run in Impact basically means that the solution loop (which is located inside smack), runs down a list of element objects and tell each of them to:

- Calculate its strains (based on the nodes displacement)
- Calculate its stresses (based on the just calculated strains)
- Calculate the resulting nodal forces and put them onto the connecting nodes

Since the element knows about its nodes and has a local material law to talk to, everything can be handled inside the element.

The abstract Element class

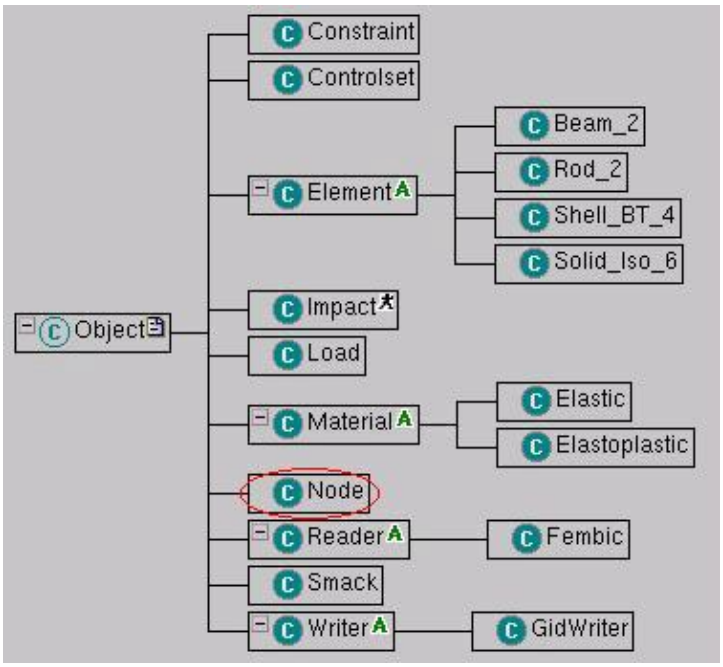
The purpose of the abstract element class (encircled) is threefold:

1. It is a framework for how a new element (sub)class should be written
2. It enables (through inheritance) all element subclasses to be treated as a single element class and thereby, no changes in the rest of Impact is needed
3. It contains some useful general methods that could be used by the element subclasses

Each element class interprets its own indata. This means that the programmer does not need to change the reader or the writer class when adding a new element class. This is handled by the actual element class itself.

More details about this comes later in the chapter about how to add an element

The Structure of the Node Class



A model consists of elements and nodes. The Node class defines the nodes behaviour.

In explicit codes, nodes are a bit extra important since the elements really only provide forces that act on the nodes.

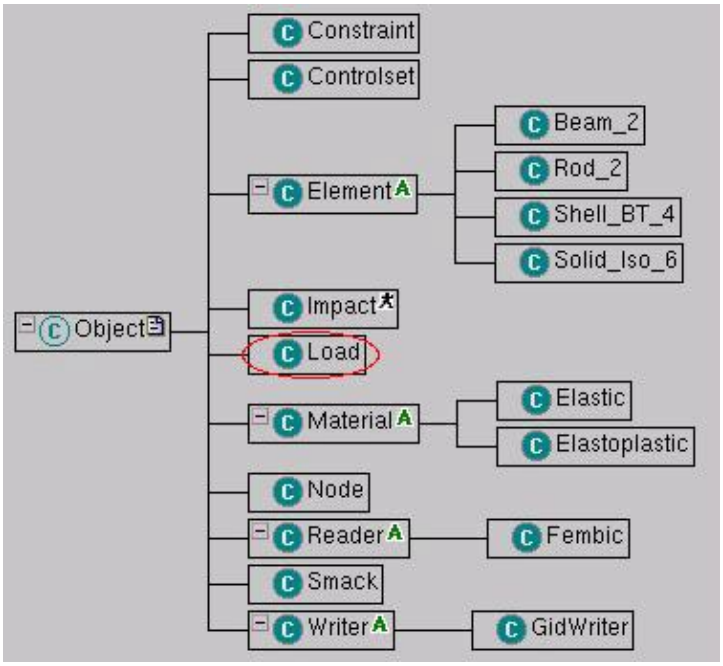
All the mass of the elements are moved out into the nodes at the start of the solution process (initialization) and from there on, the simple formula of A=F/M is used to calculate the acceleration resulting from the forces put on the nodes by the elements.

The only thing the node has to do really is to be able to calculate it's acceleration, velocity and position in space and to accept forces and mass from the elements.

In the future, contact intelligence will also be built into the nodes.

As mentioned before, the node class is used by the reader class to create one node object for each node defined in the indata file. The node object is then fed some indata like its position in space and any constraints that might be applied on it.

The Structure of the Load Class

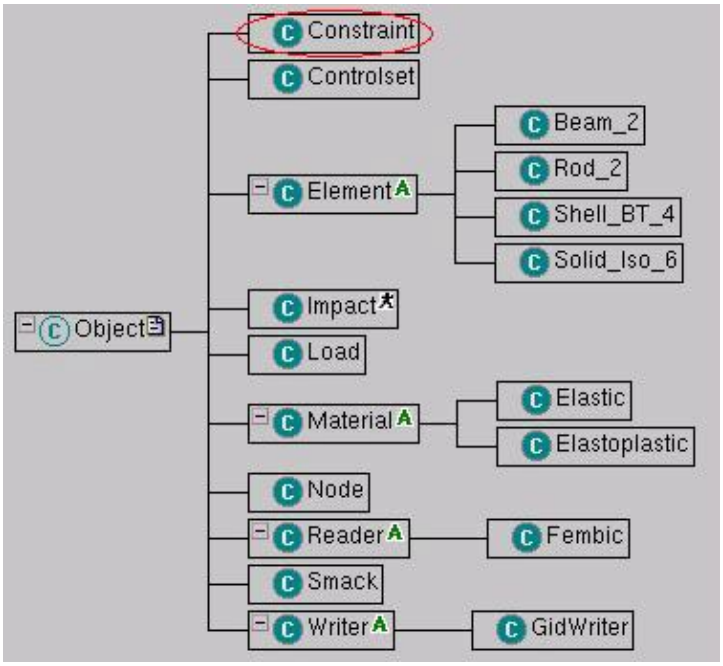


The Load class is more or less a container for data regarding the different load cases that can be applied to elements and nodes.

In the start of a run, the indata file is scanned for load sets. For each load set, a load object is created and fed with the corresponding data (example Fx = 5 kN).

Each element or node which uses this load set, then has a handle to load object and can ask it about which loads that is applied. This is used especially by the nodes to figure out which loads that has to be added before it updates it's position.

The Structure of the Constraint Class



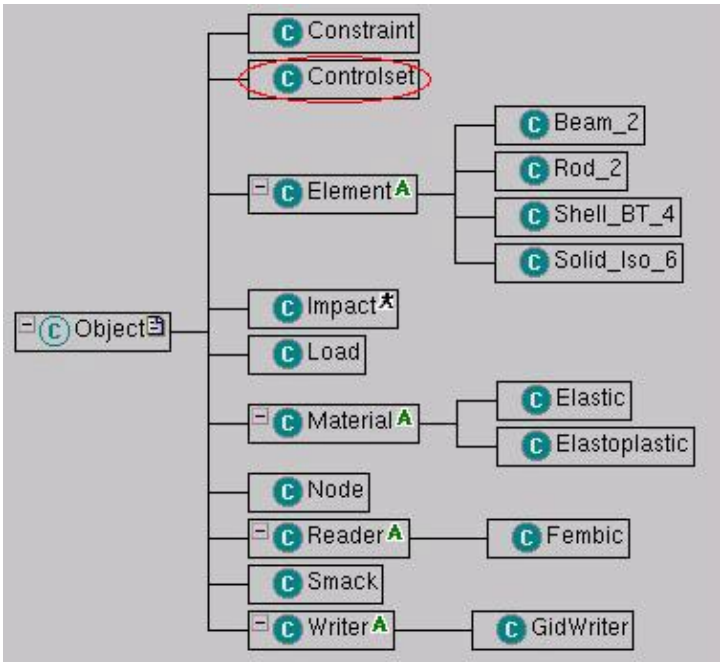
The setup for the Constraint class is quite similar to the load class.

In the start of a run, the indata file is scanned for constraint sets. For each constraint set, a constraint object is created and fed with the corresponding data (example Ax = 5 m/s^2).

Each element or node which uses this constraint set, then has a handle to it and can ask it about which constraints that are applied. This is used especially by the nodes to figure out which constraints that has to be applied before and after it has updated it's position.

There can be several different types of constraints. Therefore the constraint class is a superclass and there are different subclasses. Examples include the standard BoundaryCondition and RigidBody.

The Structure of the Controlset Class



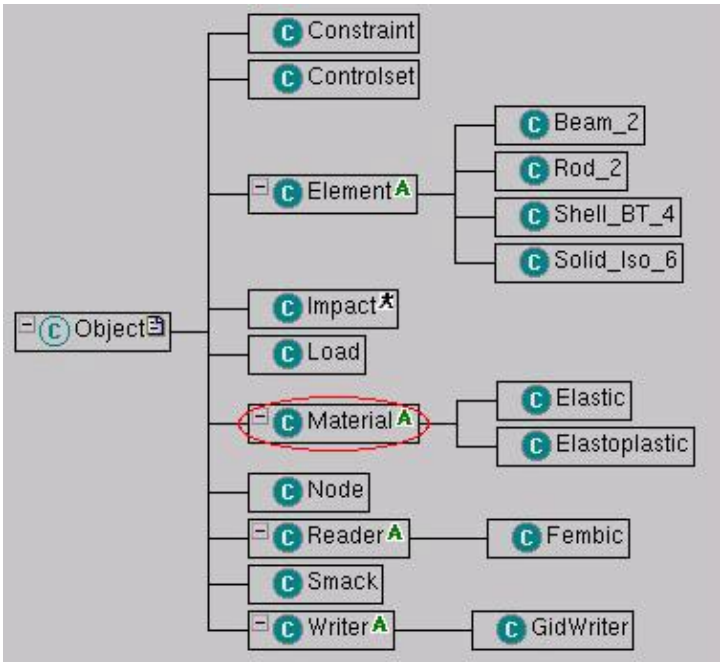
The controlset class is a special creature. It contains all the different control parameters that can be used to control the solution of a problem. Examples of such are:

- Time step size or autostep
- Start and end time for the solution run
- When to print
- .....
- .....

At the start of a run, the indata file is read (by the reader) and a controlset object is created and fed with all the control data. Note that there can only be one control set object for each solution.

The controlset object is used by the smack object to control the solution. The main solver routine inside smack asks the control object about all the things in the list above so that the solution is performed correctly.

The Structure of the Material Class



This class is the abstract "mother" class for all the different material laws that can be defined. The basic function of a material law is to calculate stresses from a given strain matrix and a given strain increment (also in a matrix format).

For a purely elastic material law, this is quite simple. Stress = Young's_modulus * strain, if we think one dimensional. When we move into elasto−plastic material laws, life becomes a little bit more complicated, but the basic principle remains. Return a stress, given a set of strains.

For reasons described in the element class, a local material object is created and embedded into each element object that is created. This material object is then "alive" throughout the whole solution and is having a frequent communication with its "mother" element, all the time answering the same question (about the stress).

The purpose with the abstract material class (encircled) is threefold:

1. It is a framework for how a new material (sub)class should be written
2. It enables (through inheritance) all material subclasses to be treated as a single material class and thereby, no changes in the rest of Impact is needed
3. It contains some useful general methods that could be used by the material subclasses

At the current state of writing, there are only two material subclasses defined. One elastic material law and one elasto−plastic.

Each material law class interprets its own indata. This means that the programmer does not need to change the reader or the writer class when adding a new material law class. This is handled by the actual material class itself. More about this can be read in the chapter about how to add a material law sub class

The structure of the tracker class

The tracker class has been carefully designed to be self containing and easy to expand. The problem when writing a program is usually that when a programmer wants to add another feature (such as a new tracker type), it also means digging around in a lot of code which is indirectly related to the new tracker type.

The benefit of Impact is that this is not the case here. The programmer only needs to consider which input syntax the new tracker should use. Once that is decided, the programmer is then free to add the new tracker along the lines described later in this manual. The basic method is to add a new subclass to the existing abstract class "tracker" and then fill that new subclass with code.

When Impact is run and a number of trackers are read in from the input file, a corresponding number of tracker objects are created. One object for each tracker.

The object will get a "memory" where it remembers which nodes or elements it should read results from and so on.

Other solutions

A Solution run in Impact basically means that the solution loop (which is located inside smack), runs down a list of tracker objects and tell each of them to:

- Collect data from the nodes and/or elements.
- Calculate the output.
- Print the output to a specific file

Since the tracker knows about its nodes, everything can be handled inside the tracker.

The abstract tracker class

The purpose with the abstract tracker class is threefold:

1. It is a framework for how a new tracker (sub)class should be written
2. It enables (through inheritance) all tracker subclasses to be treated as a single tracker class and thereby, no changes in the rest of Impact is needed
3. It contains some useful general methods that could be used by the tracker subclasses

Each tracker class interprets its own indata. This means that the programmer does not need to change the reader or the writer class when adding a new tracker class. This is handled by the actual tracker class itself.

More details about this comes later in the chapter about how to add an tracker

The Structure of the TrackWriter Class

The TrackWriter class is the mother class of all the different TrackWriter classes. Each subclass provides a different output format. An example of such a subclass is the GidTrackWriter class which prints results so that they can be read by the GID pre− and postprocessor.

A trackwriter object is created at the start of the solution run and it is then called every time that Impact wants to print to an outdata file. This can happen several times during a solution. The TrackWriter class then walks through the database of elements and nodes and asks each of them to write their status.

An equal challenge as for the reader class is present here regarding being able to add new element classes without having to go into the trackwriter class and change. The solution is the same, namely that the trackwriter class asks the element object to print it's own line of outdata (in the right format), which is then directly written to the outdata file. All the preparation is made inside the element itself.

Due to how the outdata file is structured, the method that writes the data in the element is somewhat complex. The trackwriter calls the same method several times, depending on if it wants the element to write data for a header or for another location in the outdata file. This is accomplished using control codes and is more explained in the Tracker chapter.

This layout means that the tracker class needs to have one method for each new TrackWriter class (output file format), but since the amount of file formats are significantly less than the amount of tracker types, this is feasible.

Contacts

One of the most challenging parts of a dynamic finite element program is to design a simple and robust contact searching algorithm. The following chapter describes the one used by Impact

The concept

There are two major classes of contact algorithms. One searches nodes and checks them against a surface to determine if contact is imminent. The second class searches element surfaces and checks them against other surfaces (and edges) to determine shortest distance and thereby if contact is imminent.

The element to element concept is a bit trickier to implement since we want a concept that is as general as possible and not dependent on any element shape or type. The node to element type is simpler in design and well documented which makes it the Impact choice for now.

The distance from a node to a surface can be tricky to calculate if the surface is complex. If, however, the surface is a simple triangle, it is just a matter of linear algebra. We also want a general method which works for different type of elements. These two reasons has resulted in a concept where contact "finite surfaces" are used.

Any element uses one or several contact surfaces to handle the contact with other elements (or rather nodes). When the element is asked to check its contacts, the element in turn asks each of its surfaces to check for contacts. The resulting contact forces are then distributed onto the element nodes. Each element handles its own surfaces.

An example could be the BLT Quad shell element already implemented in Impact. Since this element is flexible and can bend, two flat triangular contact surfaces cannot accurately model the shell element. In this case, a number of four would be suitable. Each of them connected to a centre point on the shell element and to two corner nodes respectively. It would represent the curvature in a rough but simple way and even allow for self contact, i.e. when the element wraps around and contacts itself. This can happen for example in a fold of a member in crush.

The beauty with this approach is that every new element can use the suitable configuration of these surfaces thus allowing a very general implementation. The logic of contact search etc is all isolated and put inside the contact surface "element", making it easy to upgrade in the future.

Nodal Search

Each element (and contact surfaces included in that element) has so search the solution space for nodes which can be in contact. For each node, it has to calculate the distance from the surface and determine if contact is imminent. Imagine now, that you have 100000 nodes in the model you want to solve and almost as many elements and it is easy to conclude that searching each node against each surface for each timestep, can take a considerable amount of time. Clearly, some kind of limitations on how many nodes to search must be made. One method, implemented in Dyna (another code) is to use so called bucket search. This means that the solution space is divided into buckets and the element residing inside a certain bucket will only check the nodes inside that bucket and neighbouring buckets. This approach is brought even further in Impact and has in fact been considered from the start.

Due to the object oriented approach in Impact, it is natural that each node should have a handle to the closest neighbouring node on both sides. This handle is constantly updated for the node when a new position is calculated. The "neighbour" knowledge of the nodes comes in handy now when nodal search must be done. Put simply, the element in question only needs to search the nodes in its immediate vicinity and that means searching all the nodes between it,s own nodes since these are on the edge of the element.

The neighbour search works only in one direction. Currently, this is programmed to be the global x−direction in the solution space. That means that any element must start with the one of it,s nodes with the smallest x−coordinate. Next, ask that one which neighbour node it has with higher x−coordinate. Check this node for contact. Next, ask that node for it,s neighbour and so on until it reaches that of it,s own nodes with the highest x−coordinate.

The amount of nodes to search now becomes much smaller than before and can usually be even smaller if the model is oriented optimally in the x−direction. For the shell element in this example, four contact surfaces needs to be checked against each node. All of this coding is isolated inside the element. The only command given to the element is to check for contact.

The Implementation

Contact handling in Impact is implemented through the use of contact elements. Contact sensing against surfaces are handled by the Contact_Triangle element and sensing against edges or beams are handled by the Contact_Line element. The implementation of these elements will now be described.

Surface contact

The basic element to sense surface contact is the triangle. A triangle defines a plane on which the node to be checked for contact can be projected. The process looks like this:

1. Use the three corner nodes to determine the plane of projection
2. Loop through all nodes between the corner nodes, starting with the node with the smallest x−coordinate
3. Check if the node is close enough in Y and Z to be in contact range.
4. Project the node onto the plane
5. Check if the node is within the triangle surface
6. If it is, calculate the distance from the plane
7. If within defined thickness, calculate the repelling force
8. Apply the force onto the node. Apply reaction force onto the element nodes
9. Proceed to check the next node until the other end node of the element is reached (the one with the largest x−coordinate).

The Contact_Triangle element can be either used on its own by the user, just like an ordinary element, or embedded into another element to provide contact handling by default.

Embedding

Embedding is a simple way of providing contact handling to your element. One example is the Shell_C0_3 element which uses one Contact_Triangle element. As the indata file is read in the Parse_GID method, a Contact_Triangle instance is created (unless the user has chosen not to). This instance is immediately fed the appropriate input into it's own Parse method, given the thickness of the element (which is also the contact thickness) and the shell elements own nodes.

The only thing remaining now is to run the appropriate methods on the contact element as the various methods of the shell element are run. For example, as the calculateContactForces method is run, the shell element only executes the same method on the embedded contact element. Some elements may require several embedded contact elements and it is up to the programmer to make sure they are all executed in the right manner. Have a look in the Shell_C0_3 element to see how the implementation has been made.

Contact_Line elements

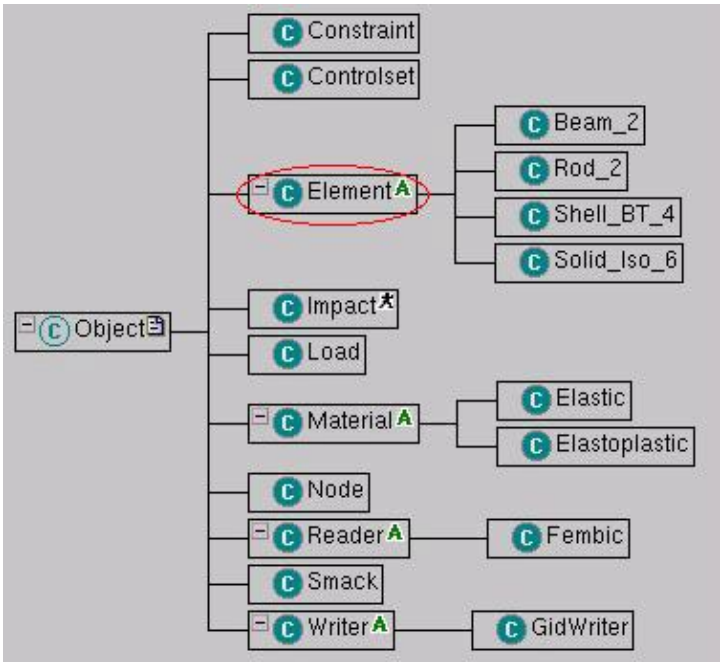
The Contact_Line element is used to detect contacts between line segments. This type of contact is useful in rods, beams and edges of shell or solid elements. It differs a bit from the surface to node contact in that contact checks are only made between line elements and nodes are not involved. This makes the scan somewhat more complicated and also more difficult to get computationally effective.

The algorithm is as follows:

1. Loop from the node with the smallest x−coordinate to the node with the largest x−coordinate
2. Check if the node has any contact_line elements attached to it.
3. If so, loop through these elements one by one and..
4. Check if parallel (then do a different evaluation)
5. If not, calculate contact point and distance
6. Apply contact force to element and reaction force to itself
7. End loop of elements
8. End loop of nodes

Similar to the Contact_Triangle element, the Contact_Line element has no stiffness and can therefore be embedded in various elements. The element has a contact diameter. Any other contact_Line element coming within this diameter will sense contact.

Creating a New Element Type



The Element is the main part of a finite element program. It is here that the program spend about 80% of it's time during calculation, but when it comes to Explicit Finite Element codes, the task that the element should perform is quite simple.

One can say the the task of a finite element is to do the following:

1. Use the position of the nodes that the element is connected to, to calculate the strain in the element.
2. Use this strain and the material law that the element is supposed to use, to figure out the resulting stress in the element.
3. Use this stress and the geometry of the element to figure out what the resulting reaction loads on the nodes will be, and apply them to the nodes.

These three tasks are repeated over and over again in a simulation, once for every timestep. If an element has more than one integration point, the process has to be repeated once for every integration point as well. This means that the calculation for an 8−integration point solid element is 8 times longer per timestep, than for an 1−point solid. This is also the reason why some fancy techniques such as hourglass control have been developed to increase speed and allow use of fewer integration points in elements.

If you want to add a new element, the challenge will be to find the formulas you need to calculate the above tasks and then implement them. The framework on how to do this is already within Impact and starts with creating a new element type. It means creating a new sub−class to the Element class. Examples of sub−classes are the Beam_2, Rod_2, Shell_BT_4 and Solid_Iso_6 classes

Each subclass has the same set of methods (which are decided by the Element class), but the coding is very different. We will investigate each method later on in this chapter.

If you want to add a new element type, the easiest way is to pick a type which can use the currently available material laws. At the time of writing the material laws can handle:

- One dimensional stress
- Plane stress
- Three dimensional stress

The three dimensional stress is used in solid elements while the plane stress is used in shell elements. One dimensional is used in the Rod element.

You can read more about this in the material chapter

The Element Class

The Element class is the mother class for all the different element types. This class does not actually contain any code, apart from a number of general methods which can be used by the sub−classes. The rest of the methods are abstract methods. This means that they act as a framework for the subclasses and if you don't include each of the methods in your sub−class, you will get an error when compiling.

The rest of Impact relies on that your new sub−class has code for each of these methods. If you don't, you will not be able to solve problems with your element.

Creating a Sub Class

Creating a sub−class is simple, but you should give it a proper name. If you look at the names currently defined, they reflect:

- Type of element (beam etc.)
- Number of nodes that the element has

When you create a new sub−class you must update the list of elements in the **getElementOfType_Fembic** method and if more file formats exists also in these related methods.

Each element sub−class also has a unique Type id, stored in the Type field. The syntax for this parameter is a three digit as follows:

First digit

- 1 − Point elements (just one node)
- 2 − Linear elements (Rod and Beam)
- 3 − Triangular elements
- 4 − Quad elements
- 5 − Tetrahedra elements
- 6 − Hexahedra elements

The second digit is a group number (ex. rod has 0 and Beam has 1)

The third digit is an index. If there are several different rod types this is used to separate them

The Methods

The Methods in the Element class are either Abstract or General. General means that they are supposed to be used by all the element sub−classes, but the coding resides in the element class since it does not change from sub−class to sub−class.

The general methods in the Element class

The general methods are automatically inherited and can be used by all sub−classes. Here follows a description:

getElementOfType_Fembic

This method is used by the reader class to generate elements based on how they are described in the indata file. The reason for putting this method here and not in the reader class is once again to keep all the related work of adding an element type, strictly within the element class. Therefore, the reader will use this method to determine which element object to generate.

The _Fembic part of the name is related to the fact that the Fembic type indata format is assumed. If a new indata file format is added, i.e. a new reader sub−class is written, an extra method will be required here, which is tailored to that file format.

calculateLocalBaseVectors

Some elements use local coordinate systems to make calculations simpler. Examples include shell elements. This method can be used to calculate the transformation matrix that transform from local to global coordinates system. The input are three nodal coordinates which form the coordinate system axes. See the code for more detailed explanation

findMaterial

This method searches the database and returns a handle to the material object with a specific name. It is used by the parse method in the element sub−classes to interpret element indata.

findNode

This method searches the database and returns a handle to the node object with a specific number. It is used by the parse method in the element sub−classes to interpret element indata.

getNodeNumber

This is quite a useful method. When you have a string with a range of numbers that are separated by commas, this method picks the n:th number in this string, converts it into an integer and returns it. It can be useful when reading indata and is used by the element sub−classes in their parse method.

isProcessed

Sometimes it can be useful to know if an element has been asked or not. The parameter 'processed' can be set and read to determine just that. This method returns the value of that parameter and since it is inherited to all sub−elements, it is placed here in the mother element class.

setProcessed

Allows you to set or clear that parameter mentioned above.

The Abstract methods in the Element class

The Abstract methods are empty in coding in the element class. This is because every sub−class has a different implementation of these methods. As a programmer that wishes to add a new element sub−class, these methods are the ones that he/she has to write. Here follows a description of each method

assembleMassMatrix

This method calculates the elements contribution regarding mass and rotational inertia to the nodes and is called at the initialisation of the problem. Since this program uses lumped mass, all the mass are concentrated to the nodes, and so are the inertia of rotation as well. The mass is usually quite simple to calculate. In the example of a rod, it is half of the total mass of the rod in each node.

For a shell element it could be slightly more trickier. The difficult part comes when the rotational inertia is to be calculated since this is needs to be calculated in three dimensions and then transformed to the global xyz coordinate system before adding it to the node.

Roughly, the procedure is as follows:

1. Calculate element mass distribution and add to node
2. Calculate element inertia
3. Transform to global directions
4. Add to node inertia matrix

calculateContactForces

The contact forces are calculated by this method. Contact forces are forces that are generated onto nodes because they are about to contact an element. To prevent the node to just pass through the element, an artificial force is calculated and put on to the node (with the opposite force put onto the element).

This is done in the following way:

1. Let the element check it's segment(s) or faces against any neighbouring nodes.
2. calculate distance from segment to node.

3. If distance is smaller than tolerance then calculate a reaction force as a function of the distance.
4. If a reaction force has been calculated, then add this force to the node and the opposite force to the elements nodes.
5. Continue this loop until all the nodes within the contact tolerance from the element has been checked.

It is worth noting that there are loads of different contact algorithms out there. Each of them with different benefits. The good thing with this program is that each node knows about it's closest neighbour node! This can be used to limit the amount of nodes you need to check for your contact algorithm.

After each timestep, the nodes will update themselves to see if their closes neighbour has changed.

calculateExternalForces

This method computes the contribution from gravity and external forces on the elements. The gravity and external forces on the elements are transformed into nodal loads which are then added to the nodes connected to the element.

calculateNodalForces

This method calculates and adds the element internal forces to the nodes. Please note that by definition, the internal forces are subtracted from the external loads. Thus, the "addition" is negative.

The coding can be quite extensive in this method for some elements. In principle, the approach is however more or less the same. If we look at the rod, the coding is quite simple:

Start by calculating the local force using stress in rod and cross sectional area

```
force.set(0, 0,stress.get(0,0)*cross_section_area);
force.set(1, 0, 0);
force.set(2, 0, 0);
```

Now, continue by transforming this force to global coordinates using a transformation matrix calculated in another element method.

```
global_force = local_coordinate_system.times(force).copy();
```

Finally, add this force contribution to the nodes. Remember to use negative addition since this is an internal force (a reaction force).

```
node1.addForce(global_force.times(1));
node2.addForce(global_force.times(−1));
```

calculateStrain

The strain of the element is calculated here. It is a matter of transforming the positions of the nodes that the element is connected to, into a strain matrix for the element.

In addition, the strain increment is also calculated (that is the difference in strain between now and the strain in the previous timestep)

Calculating the strain for the rod element is quite simple. Start by calculating the length of the rod.

```
xpos1 = node1.getX_pos();
ypos1 = node1.getY_pos();
zpos1 = node1.getZ_pos();
//
xpos2 = node2.getX_pos();
ypos2 = node2.getY_pos();
zpos2 = node2.getZ_pos();

new_length = java.lang.Math.sqrt((xpos2 − xpos1) * (xpos2 − xpos1) + (ypos2 − ypos1) * (ypos2 − ypos1) +
(zpos2 − zpos1) * (zpos2 − zpos1));
```

The strain increment can now be calculated. Note that his is true strain.

```
dstrain.set(0,0,Math.log(1+(new_length−initial_length)/initial_length) − strain.get(0,0));
```

Finally, calculate the cross section area of the rod, assuming incompressible material.

```
cross_section_area = initial_cross_section_area*initial_length/new_length;
```

The strain matrix will be updated when the material law is used to calculate the stress.

calculateStress

This method calculates the stress matrix of the element out of a strain matrix and a strain increment matrix. The main work here is actually done inside the material law of which the element uses. Which law to use is assigned to the element in the initialisation stage.

This is the code in the rod element. The stress matrix is supplied as an indata, but is actually updated inside the material law. Note that for the rod element, the one dimensional law is used. This has to be implemented for each new material law that is added.

```
material.calculateStressOneDimensional(strain,dstrain,stress);
```

checkTimestep

For each element there is a lower limit on the critical timestep. This timestep is decided by the shortest travelling distance a wave can take through the element on one step in time.

For a rod element, this is the length of the element, divided by the wave−speed of the rod material. For a hexahedron (solid) element, the critical length is the shortest side of the element and thus, the timestep is this critical length divided by the wave−speed of the material that the solid is made out of.

The material law supplies the wave−speed. This method has input and if the element has a smaller timestep than the input value, the element should return it's own timestep value.

This method is called by the main routine in smack. The program is asking all elements for their timestep before it takes it's next step. The size of this step is set by the element that requires the smallest timestep. It sets the stability of the solution.

getNumber

This method simply returns the number of the element. This number is the one that was defined for the element in the indata file.

getNumberOfIntegrationPoints

A certain element type can have several different number of integration points. An element with many interaction points has greater accuracy and stability but takes longer time to calculate.

As an example, the solid element can either have one interaction point or eight. The one integration point version is not stable since so called hourglassing can occur. There are stabilisation routines available that can fix this but at the time of writing, there is none implemented.

getType

Each element is assigned a type number (see getElementOfType_Fembic above). This method returns this number.

parse_Fembic

As described before, some of the interpretation of indata files has been put into the element itself. This method is called when a Fembic indata file is read and the indata for the element needs to be read and interpreted.

The beauty with this approach is that the element can read the indata string and set all its variables by itself. The variables can then be kept private and encapsulated inside the element.

Another benefit is that the programmer only needs to work in this class when he/she adds a new element. There is no need to fiddle around els where in impact.

When a new file format is added, a new additional method is required for this file format and the programmer needs to add this new method to all element types in impact.

print_Gid

This method is the opposite of the parse_Fembic method described above. It reads all the parameters in the element and creates an output string which is then printed into the outdata file. This very method prints the outdata from the element in GID format. GID is a pre− and postprocessor which is free and a good complement to Impact.

This method is called every time the writer object wants to print something, but there could be different things from time to time like header data or bulk data. A control parameter is supplied with the call to signal what is to be printed.

setInitialConditions

This is the big initialisation method for the element and it is called once at the beginning of the solution. Examples of what this method should do is:

- Make a local copy of the material object.
- Tell the material object to initialise itself.
- Initialise matrices that are needed for later calculation.
- Other things that may need to be done before calculation starts.

If we look at how the material object is initialised, here is example code:

```
try {
material = (Material) material.clone();
} catch (CloneNotSupportedException e) {
System.err.println("Object cannot clone");
}
```

The material object is told to clone itself and the handle is reinitialised

setNumber

This method simply assigns a number to the element.

updateLocalCoordinateSystem

Some elements has a local coordinate system. The system is really a transformation matrix between the local and global coordinates system and needs to be updated between each timestep since the element deforms and the coordinate system deforms with it.

Here is a code example. The method 'calculateLocalBaseVectors' in the Element mother−class is used and is supplied the element node coordinates.

```
local_coordinate_system = new
Jama.Matrix(super.calculateLocalBaseVectors(node1.getX_pos(),node1.getY_pos(),no
de1.getZ_pos(),node2.getX_pos(),node2.getY_pos(),node2.getZ_pos(),node2.getX_pos
```

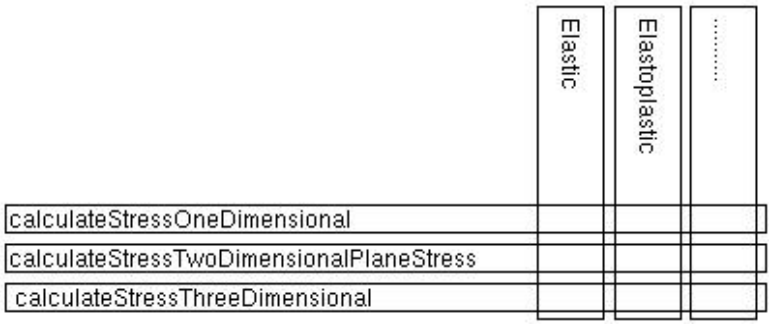
()+1,node2.getY_pos()+1,node2.getZ_pos()+1).getArray());

Other things to think of

Have a good look at the coding of the Rod_2 and the Solid_Iso_6 element. The code is full of comments. Good luck!

Creating a New Material Type

A material law reflects the behaviour of a certain material. Steel behaves different from clay and so on. To be able to model the material behaviour correctly, a material law is needed. The law "rules" how the stress response should be based on a strain input. Some laws depends on the history of the material i.e. if the material has been stretched before, it may behave differently if one tries to stretch it again. For the material law to reflect this, it needs to have a memory which makes it ideal to make the law into an object which is put inside each element that uses it. You can read more about that under the chapter adding an element.

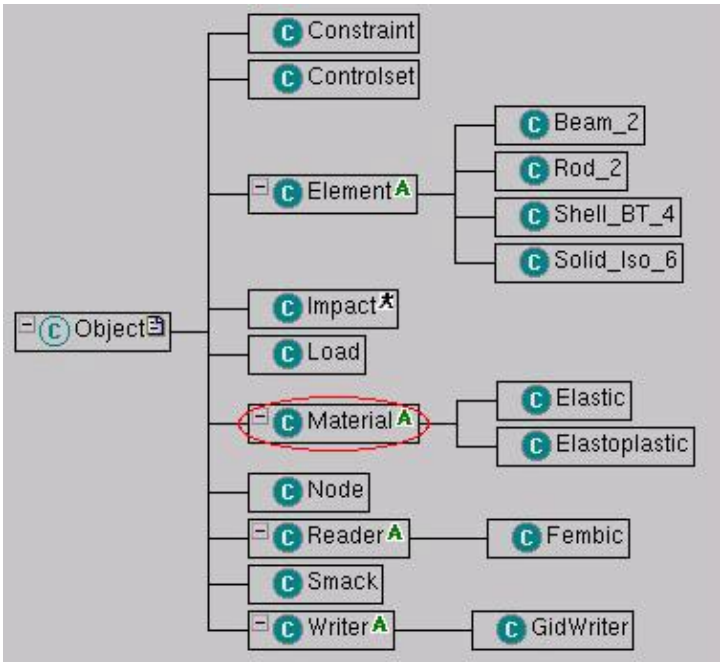


The material law has only one purpose; To determine the correct stress matrix based on a strain matrix and a strain increment matrix. There are several versions of these matrices where the most general one is of course the three dimensional one. It is used in the hexahedron element for example.

Other versions are plane stress matrix which is used in shell elements and a single one dimensional matrix which is used in a rod element. For the material law to be able to calculate all of these cases, the material law class (which is where the material law code resides), has several methods as can be seen in this figure:

The methods has to be present in all the different material law classes and be called the same for the concept to work.

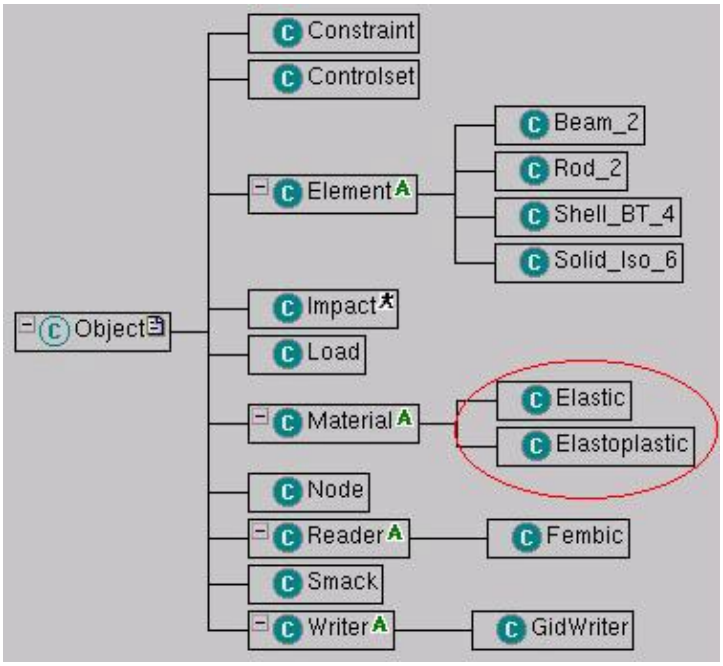
The Material Class



The material class is the mother class of all the different material laws. This class contain a large amount of abstract methods which are the ones you have to define in your subclass, should you choose to add a new material law. It also contains a few general methods which are usable for all the material subclasses. The methods will be listed in the later subsection.

All the code in Impact is written in such a way that it treats all material laws as just instances of the Material class. This is a well known concept that can be used in object oriented programming. It allows the programmer to extend Impact with new material laws without having to change code in other parts of the program. If you want to add a new material law, you just have to create a new material sub class and fill in the code in the methods that are declared in the mother class as abstract methods.

Creating a Sub Class



If you compare the methods in the mother material class and the methods in any of the sub classes, you will see that they are exactly the same apart from the code inside the methods. As an element is created (in a problem that is to be solved), a material object is also created and put inside the element. That material object is an instance of the material law subclass that was chosen for that very element in the problem indata file.

During the simulation, the element will call on the material object over and over again (once every timestep). Basically, the question will be the same, namely "here is my strain matrix. What is my state of stress?" The material law will then do some calculations to find this out and return the stress matrix to the element.

In the picture there are two subclasses defined. The elastic subclass is very simple since it only multiplies the strain with Young's modulus and returns the stress. For two and three dimensions you have some more calculations to do but the concept is the same. The elastoplastic law however is more advanced. The material behaves like in the elastic material law up to the yield stress of the material. After that level is passed, the material starts to behave like plastic material (similar to a gum) and the strain will be irrecoverable. That means that the strain will not go back to zero again should the stress drop, or seen in another way, the stress will not be zero if the strain goes back to zero. This material law has a memory. It remembers if it has passed yield stress before and even remembers the maximum stress level it has reached.

Other more advanced laws that can be added are not only depending on the strain and the loading history, but also the speed at which the material is extended, so called strain velocity hardening. Most steels constitute a significant strain velocity hardening component (up to 25% higher stress level at rapid loading).

The Methods

Now follows a description of each method in the material law. We will start with the general methods that are defined in the mother class and which are usable for all material classes.

The General Methods in the Material Class

clone

This method will clone the material object i.e. make an identical copy. It is used my the elements to create a local copy to embed inside themselves for the duration of the solution.

getDensity

This method returns the density of the material

getName

This method returns the name of the material

getNumber

This is a useful method for getting a number from a string containing several number separated with commas. It is useful in the parsing stage.

getYoungsModulus

Returns the Youngs Modulus of the material.

numberOfPoints

Returns the number of data points in an indata string by looking at the number of commas used for separation. Used in parsing of indata.

setName

Sets the name of the material object

The Abstract methods in the Element class

These methods need to be coded in the subclasses. The coding will be unique for each new material law.

calculateStressOneDimensional

This method calculates a stress given a strain and strain increase since the last timestep. In this case all matrices are assumed to be of the standard 6x1 type.

Strain		Stress
Epsilon XX		Stress XX
Epsilon YY		Stress YY
Epsilon ZZ		Stress ZZ
Gamma XY		Stress XY

Gamma YZ		Stress YZ
Gamma ZX		Stress ZX

The stress is given as an input to the method, but is changed inside the method to the return value.

calculateStressTwoDimensionalPlaneStress

Similar to the above method but for plane stress situation (used by shells)

calculateStressThreeDimensional

Similar to the above method but for three dimensional stress (used by all solid elements)

parse_Fembic

This is the method where the material law object will interpret and set it's indata. The data is taken from the indata file by the Reader object and sent here for interpretation as a string.

To parse this string you can use the methods numberOfPoints and getNumber explained above to sort out the data.

If more indata formats are added in the future, there will be more methods like this to define when you add a new material law

setInitialConditions

In this method you check that all variables has been read in and set. You also set the ones you need to set before solution starts.

wavespeedOneDimensional

Returns the speed of waves through the material. This is for one dimensional situations and is used by rod elements to determine smallest timestep.

wavespeedThreeDimensional

Returns the speed of waves through the material. This is for three dimensional situations and is used by solid elements to determine smallest timestep.

wavespeedTwoDimensional

Returns the speed of waves through the material. This is for two dimensional situations and is used by shell elements to determine smallest timestep.

Other things to think of

Keep your variables private inside your class to prevent others from fiddling about with them in ways you didn't expect

Read the code. There are plenty of documentation

Go for it! Good luck!

Adding a New Input File Format

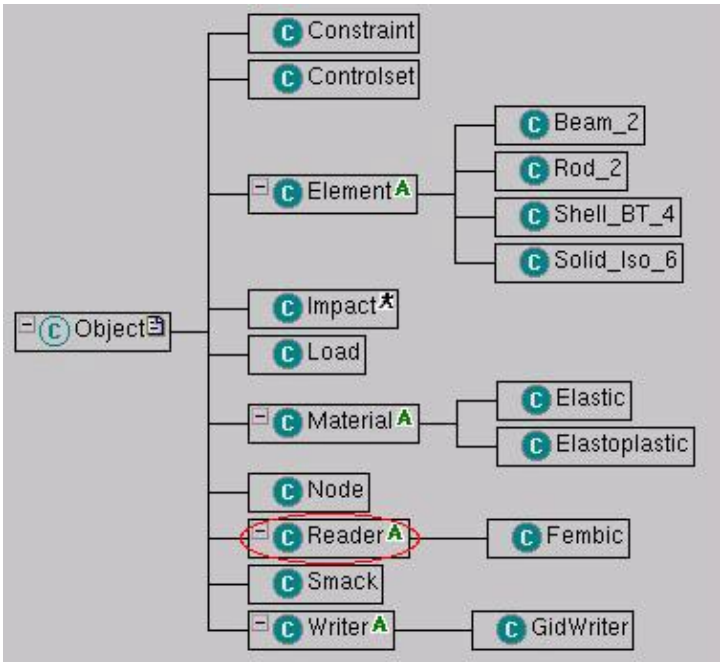
Impact is designed to be able to handle several different indata formats. For explicit finite element solvers, there are currently several different formats available and they are all connected to one of the commercial programmes. Examples are:

- LS−Dyna 3D
- Radioss
- Pam−crash

These formats all date back some time and has evolved. Recent advances include free format syntax. They all include a large amount of parameters and will require some work to understand. For this reason, Impact comes with it's own indata format, called Fembic (Basic Finite element = FemBic).

The objective of Fembic is that it must be readable and logical in format but most of all, simple! Never the less, impact is designed to be extendable to read the other formats as well and to do this, the reading and parsing of the indata files has been given to the Reader class. The design of this class and how to extend it will now be explained.

The Reader Class



The reader class is the mother class for all the different indata file readers. The Fembic format has it's own subclass to the Reader class. Impact is written so that all the code uses the reader class when a data file is to be read and parsed. In reality however, it will be one of the sub−classes that will do the job but it does not matter since the right routines will be run in any case. The benefit is that as a programmer, you do not have to worry about the rest of the program when you change something in your class, or when you add a new class. This is a very nice feature of object oriented programming.

The reader class is an abstract class. It contains very little code. It does also contain several methods which also are abstract and contains no code. They are important to understand however since you have to write each of them in your subclass and fill the method with your code.

Impact assumes that every problem can be described in the following building blocks:

- Elements − The elements that make up a model
- Nodes − The nodes that connects the elements
- Material − Description of the material that the elements are made of.
- Loads − Loads on nodes or elements
- Constraints − Constraints on elements or nodes behaviour
- Controlset − Control commands on the solution process.

The reader class therefore has methods that reads and interprets these blocks from the indata files. If an indata file is structured differently in any given format, the challenge for the reader class (or rather the specific sub−class) will be to restructure the data to fit into these blocks. They are described a little later.

Creating a Sub Class

When you want to add a new reader for a new file format (such as Dyna), you have to create a new sub class to the reader class. As you can see, there is already one called Fembic (for the Fembic file format). In this case we would call it Dyna. Now, you have to fill this class with methods according to the methods defined in the Reader class. Next you have to fill each method with the appropriate code.

As you will see later, you will also need to add methods in the element class, material class, and all their subclasses.

The subclass is initiated by the Smack object. You will also have to modify that class to make it aware of the fact that there is one more file format to select reader for. After initiation, Smack will start by asking the reader about how many of each indata blocks there are.

Knowing the amount of blocks, Smack will then continue to ask the reader for elements, nodes etc as many times as there are definitions in the indata file. Smack will handle the opening and closing of the indata files automatically.

The Methods

Methods are defined in the Reader class. There are general methods which automatically are inherited to the sub−classes and are useful additions. There are also abstract methods which the programmer has to generate in his/her's subclass and fill with appropriate code, specific for the file format to be read by that sub−class method. A description of each method will now be done:

Abstract methods

numberOfConstraints

Scan the file and return the number of constraints that are defined

numberOfControls

Scan the file and return the number of controls that are defined

numberOfElements

Scan the file and return the number of Elements (of all different types) that are defined

numberOfLoads

Scan the file and return the number of loads that are defined

numberOfNodes

Scan the file and return the number of nodes that are defined

numberOfMaterials

Scan the file and return the number of materials that are defined

getControlset

This method is called from the Smack object. It is called during initialisation of the problem and Smack will start by creating a controlset object. This object is then used as indata to the method.

The method takes the controlset object and feeds it all the control data defined in the indata file. This means reading one or several lines of the indata file. Each line must be parsed and then fed to the object. After completion, there is no need to return anything since the object already was created in the smack object.

Parameters in the control set are things like start and stop time for the solution. Time step size (or autostep). When to print.... This method is only called once since there can only be one controlset object in a solution.

getNextConstraint

This method is called several times. One for each known constraint. Each time, a constraint object is created and is then fed with parsed data from the indata file. The parsing is done in this method.

The method assumes that the file already is opened. The reason for this, is that since this method is called several times, it is important that the same constraint is not read twice. The pointer that decides which part to read, is reset when the file is closed, so that cannot happen between each call to the method.

getNextElement

Similar to the other method, it is assumed that the file already is opened. This method is called once for each

element that is to be parsed.

Since there are several different types of elements the type of element must be known before creation of the element object. The creation is made in this method and the object is then fed the rest of the indata string defined in the indata file.

Let's illustrate this in an example. The indata file starts with a block header describing a block of elements of type rod. The reader object must now remember this since the coming element will all be of this type. The next line then describes all the data for the first element. The reader creates an element object of the right type and gives the string of the indata to that element object **to parse itself.**

The element method used to parse the data for a Fembic indata file is called `parse_Fembic` For another file format, there will be another method in the element.

The problem now is that for each new file format, **you have to add a parse method in every element subclass.** This means additional work for you apart from writing the reader subclass.

You might ask why this is the case. The reason is that priority has been put to simplify addition of a new element type rather than a new file format. There are much more element types than file formats. Had the parsing of element data been put inside the reader subclass, the element programmer would have been forced to go digging into all the reader subclasses when all he wants is to add a new element. Doing it this way enables him to stay inside his own element subclass with all his work.

There is also another method that has to be added in the element mother class. Selection of an element type based on a name is made in the `getElementOfType_Fembic`, with a similar reasoning about the name of the method as above.

getNextLoad

This method is called once for each load in the indata file. Similar to the other methods it assumes that the file is already opened. It does all the parsing of the load object data itself and feeds them to the object. After completion, the object is returned to Smack.

getNextMaterial

This method is very similar to the getNextElement method in that there can be several different material types which forces a similar layout when it comes to adding extra methods in the material classes.

getNextNode

This method reads the indata file and creates a node object. The indata is parsed in this method. On thing to remember is that the data for a node includes references to constraints and loads. This means that the node object must be given references to the right constraint and load objects. In order to find those objects, this method has two lists included in the parameters given to the method. These lists are the constraint lists and load lists. They contain the previously defined constrain objects and load objects so that the method can search for the right constraints and loads to give to the node object.

When the definition is finished, the node object is returned to the smack object where it is inserted into the great node array.

open

Opens the indata file and sets up some parameters for the parser, such as which characters are to be treated as numbers, words and comments etc.

General methods

isAKeyword

This is a useful little method which checks a given word to see if it is a defined keyword. It is used during parsing to figure out if a new block has been reached in the indata file. The keywords are file format specific and are stored in an array called keywords[]. It is defined directly under each subclass.

close

Closes the indata file.

Other things to think of

- Keep your variables private to prevent someone else to access them in ways you did not foresee.
- Keep your code simple and clean.
- Cram it full of comments. It should be easy to understand.
- Remember that you will have to make additions in the element class, material class and all their subclasses when you add a new reader (subclass).
- Do not forget to document your reader with a chapter in this manual.
- Have Fun!

Adding a New Output File Format

Impact is designed to be able to write result data in various formats. There are several different post−processors available to view this result data and the default one for Impact is GID.

To enable writing of data, a writer object is used. The object is initiated as soon as the indata file is read because it is the indata file that decides which format to print in. If nothing is specified, the GID writer object is used.

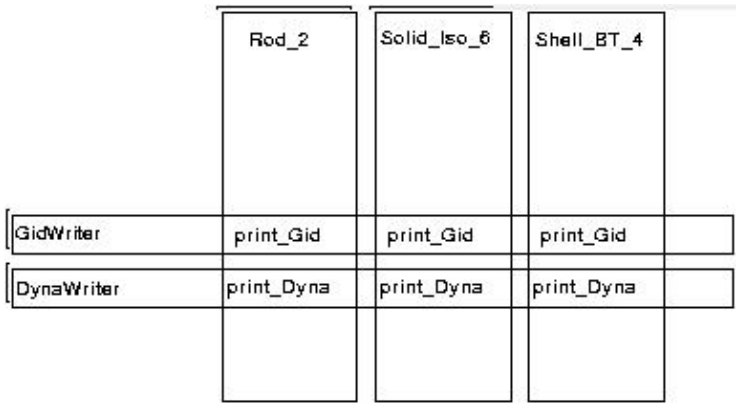
The Writer Class

The writer class is the mother class for all the different writers. Similar to the Reader class, it mainly contains abstract methods and a few general methods for the subclasses to use. They will be described later.

Creating a Sub Class

Adding a new output format means adding a new subclass to the writer class. This subclass has the task of structuring and directing the output. It works by calling the elements and nodes (which contain the results) and asks them to print their data one by one. The whole "orchestration" of this procedure is done by an object initiated from this subclass.

At the time of writing, there is only one subclass, called GidWriter. The main method for this class is the write method which is used by the Smack object to order printing of results. During the simulation, this method will be called several times since explicit programs dump their results frequently during simulation. We will go through this method in detail in the next part.



The data we want to print is all contained inside element and node objects. To make matter worse, the data is also encapsulated inside these objects in private variables, making them unreachable from our GidWriter object. The way to print this data is then to ask the elements to print it. But how does the element know which format to print the results in? The output can look very different from format to format.

The solution is to create a unique printing method for each output format. For each new output format added to Impact, it will mean that each element sub−class will have to add one more print method. (The node class

already has methods which provides the results data). This approach makes it "messy" to add new output formats since the programmer has to add code in other classes than the writer and writer sub−classes, but since the amount of output formats are far less than amount of element types, this seems to be the most rational approach.

We will soon have a look at what the extra methods in the element classes will mean.

The Methods

write

When this method is called, it is time to write the output data from the solution. The implementation of this method in the GidWriter looks as follows:

```
public void write(String fname, double currtime) {
filename = new String(fname);
time = currtime;
try {
if (counter == 0) {
writeMesh();
}
writeResult();
counter++;
} catch (IOException ioe) {
System.out.println(ioe);
return;
}
}
```

Two private methods are used here, writeMesh and writeResult. They are the ones doing the main job. The first time the write method is called, some mesh data is written through the writeMesh method in addition to some results. The following times, no more mesh data is written. This is what the GID post−processor wants as input so that is what we must write.

writeMesh

This is a private method. It may only exist for the GidWriter class but I will explain what it does in any case to give a good feeling for this concept.

The method basically gets information from the node objects by asking them for their position in space and then formatting this information to print a mesh data file in the correct format. It can ask the nodes directly since there are general methods for all the data in the node.

It also asks the elements to print all their data about which nodes each element is connected to. This is data that each element object knows, but it is private data. Have a look at this line of code:

```
bw.write(temp_element.print_Gid(Element.MESH, 0)
```

Here, the element is asked to print the data using a special control code (Element.MESH). In other cases, the element may be asked to print something else and then another control code will be used. The resulting string of data is then directly written to the file (bw).

writeResult

The principle used in this method is the same. Each element is asked to print it's data but by using a control code, the data printed by the element may differ according to need. You can find the details in the code itself.

Changes to the Element Classes

Each element must contain a method to print it's data. The name of this method is standardised to **print_*name*** where *name* is the name of the output format.

The method should access the data in the element and create an output string which will be printed by the writer object. The request from the writer may differ depending on situation and therefore a control code must also be sent which the method must follow to do the right thing

The second parameter is the gauss point number of the element. Since each element will contain at least one Gauss point and the results are stored there, printing will be made for each gauss point. This parameter specifies which gauss point to read and print.

Naturally, the implementation of this method will differ from element type to type which is why each element type needs this method. Remember now when you write a new output format to go through each element sub−class and add this method.

Other things to think of

- Keep your variables private to prevent someone else to access them in ways you did not foresee.
- Keep your code simple and clean.
- Cram it full of comments. It should be easy to understand.
- Remember that you will have to make additions in all the element subclasses, when you add a new writer (subclass).
- Do not forget to document your new writer with a chapter in this manual.
- Have Fun!

Creating a New Tracker Type

The results generated by the simulation cannot always be presented in the right way. Sometimes there is a need to "measure" the movement or force on a single node for example and this is where trackers come in handy. The measured results can be calculated upon and the written to a data file specified by the user.

There are a range of trackers available:

1. Nodeforcetracker − used to measure forces and moments on a single node
2. Sectionforcetracker − measures the force through a section by reading results from a range of nodes around the section and then summing them up
3. Sectionmomenttracker − Similar to the sectionforcetracker but reading the moment in a section.

If you want to add a new tracker, the framework on how to do this is already within Impact and starts with creating a new tracker type. It means creating a new sub−class to the Tracker class. Examples of sub−classes are the NodeForceTracker or SectionForceTracker classes

Each subclass has the same set of methods (which are decided by the Tracker class), but the coding is very different. We will investigate each method later on in this chapter.

If you want to add a new tracker type, the easiest way is to pick a similar type which already reads the kind of data you are interested in and modify it.

The Tracker Class

The Tracker class is the mother class for all the different tracker types. This class does not actually contain any code, apart from a number of general methods which can be used by the sub−classes. The rest of the methods are abstract methods. This means that they act as a framework for the subclasses and if you don't include each of the methods in your sub−class, you will get an error when compiling.

The rest of Impact relies on that your new sub−class has code for each of these methods. If you don't, you will not be able to solve problems with your tracker.

Creating a Sub Class

Creating a sub−class is simple, but you should give it a proper name. If you look at the names currently defined, they reflect what the tracker does, for example nodeforce etc.

When you create a new sub−class you must update the list of trackers in the **getTrackerOfType_Fembic** method and if more file formats exists also in these related methods.

The Methods

The Methods in the tracker class are either Abstract or General. General means that they are supposed to be

used by all the tracker sub−classes, but the coding resides in the tracker class since it does not change from sub−class to sub−class.

The general methods in the tracker class

The general methods are automatically inherited and can be used by all sub−classes. Here follows a description:

getTrackerOfType_Fembic

This method is used by the reader class to generate trackers based on how they are described in the indata file. The reason for putting this method here and not in the reader class is once again to keep all the related work of adding an tracker type, strictly within the tracker class. Therefore, the reader will use this method to determine which tracker object to generate.

The_Fembic part of the name is related to the fact that the Fembic type indata format is assumed. If a new indata file format is added, i.e. a new reader sub−class is written, an extra method will be required here, which is tailored to that file format.

findNode

This method searches the database and returns a handle to the node object with a specific number. It is used by the parse method in the tracker sub−classes to interpret tracker indata.

getNodeNumber

This is quite a useful method. When you have a string with a range of numbers that are separated by commas, this method picks the n:th number in this string, converts it into an integer and returns it. It can be useful when reading indata and is used by the tracker sub−classes in their parse method.

The Abstract methods in the tracker class

The Abstract methods are empty in coding in the tracker class. This is because every sub−class has a different implementation of these methods. As a programmer that wishes to add a new tracker sub−class, these methods are the ones that he/she has to write. Here follows a description of each method

collectData

This method is run by smack (the master class that runs the simulation) to order the tracker to ask around and collect all the needed data from the nodes or elements that it wants to measure.

calculate

The measured data is here calculated to get the resultdata for the tracker. This is usually the "heart" of the tracker.

checkIndata

This method is used to check that all the necessary data has been given to the tracker from the indata file and that the numbers given are accurate.

getNumber

This method simply returns the number of the tracker. This number is the one that was defined for the tracker in the indata file.

parse_Fembic

As described before, some of the interpretation of indata files has been put into the tracker itself. This method is called when a Fembic indata file is read and the indata for the tracker needs to be read and interpreted.

The beauty with this approach is that the tracker can read the indata string and set all its variables by itself. The variables can then be kept private and encapsuled inside the tracker.

Another benefit is that the programmer only needs to work in this class when he/she adds a new tracker. There is no need to fiddle around elsewhere in impact.

When a new file format is added, a new additional method is required for this file format and the programmer needs to add this new method to all tracker types in impact.

print_Gid

This method is the opposite of the parse_Fembic method described above. It creates an output string which is then printed into the outdata file. This very method prints the outdata from the tracker in GID format. GID is a pre− and postprocessor which is free and a good complement to Impact.

This method is called every time the TrackWriter object wants to print something, but there could be different things from time to time like header data or bulk data. A control parameter is supplied with the call to signal what is to be printed.

setInitialConditions

This is the initialisation method for the tracker and it is called once at the beginning of the solution.

setNumber

This method is simply used to assign the number to the tracker.

Other things to think of

Have a good look at the coding of the NodeForce and the SectionForce tracker. The code is full of comments. Good luck!

Adding a New Tracker Output File Format

Impact is designed to be able to write result data in various formats. There are several different post−processors available to view this result data and the default one for Impact is GID.

To enable writing of tracker data, a trackwriter object is used. The object is initiated as soon as the indata file is read because it is the indata file that decides which format to print in. If nothing is specified, the GID trackwriter object is used.

The TrackWriter Class

The trackwriter class is the mother class for all the different trackwriters. Similar to the Reader class, it mainly contains abstract methods and a few general methods for the subclasses to use. They will be described later.

Creting a Sub Class

Adding a new output format means adding a new subclass to the trackwriter class. This subclass has the task of structuring and directing the output. It works by calling the different trackers, asking them to collect and calculate the results and finally to print their data one by one. The whole "orchestration" of this procedure is done by an object initiated from this subclass.

At the time of writing, there is only one subclass, called Gidtrackwriter. The main method for this class is the write method which is used by the Smack object to order printing of results. During the simulation, this method will be called several times since explicit programs dump their results frequently during simulation. We will go through this method in detail in the next part.

The data we want to print is all contained inside tracker objects. To make matter worse, the data is also encapsulated inside these objects in private variables, making them unreachable from our Gidtrackwriter object. The way to print this data is then to ask the trackers to print it. But how does the tracker know which format to print the results in? The output can look very different from format to format.

The solution is to create a unique printing method for each output format. For each new output format added to Impact, it will mean that each tracker sub−class will have to add one more print method. This approach makes it "messy" to add new output formats since the programmer has to add code in other classes than the trackwriter and trackwriter sub−classes, but since the amount of output formats are far less than amount of tracker types, this seems to be the most rational approach.

Remember that as you add a new trackwriter type, you must add that type to the trackwriter getTrackWriterOfType_xxxx method(s).

The Methods

write

When this method is called, it is time to write the output data from the solution. The Gidtrackwriter just asks each tracker in turn to write it's result through the use of the writeResult method.

Other things to think of

- Keep your variables private to prevent someone else to access them in ways you did not foresee.
- Keep your code simple and clean.
- Cram it full of comments. It should be easy to understand.
- Remember that you will have to make additions in all the tracker subclasses, when you add a new trackwriter.
- Do not forget to document your new trackwriter with a chapter in this manual.
- Have Fun!