Command's Description

The definition of command_i which is the base class of all the commands.

```
class command_i
{
    public:
        command_i(){}
        virtual ~command_i(){}
        virtual bool exec() = 0;            // command execution
};
```

Following are the classes inherit from class command_i. These classes are created with the intention of better use of MORT. These commands will meet many kinds of requirements.

```
class addions_command : public command_i
{
    public:
        addions_command( const string& dest, const string& ion, int number );
        addions_command( const string& dest, const string& ion1, int numion1,
const string& ion2, int numion2 );
        virtual ~addions_command();
        virtual bool exec();

    private:
        string m_dest;
        string m_ion1;
        int m_numion1;
        string m_ion2;
```

```
        int m_numion2;
        int m_runtyp;
};
```

Class addions_command: Add ions to molecule “dest”. It has two constructor functions with different arguments, details of arguments can be seen in the Supporting Information, addions section. “shlext” and “res” are set to 4.0 and 1.0 separately by default.

```
class addmap_command : public command_i
{
    public:
        addmap_command(const string& type, const string& nmap );
        virtual ~addmap_command();
        virtual bool exec();

    private:
         string m_type;
        vector< vector<string> > m_nmap;
};
```

Class addmap_command: Parse the namemap part of force field. It can operate differently with two kinds of type. “type” can be set to “addpdbresmap” and “addpdbatommap”, “addpdbresmap” is used to add the relationship between residues, and “addpdbatommap” is applied to atoms. All the information interpreted is stored in **content()** with a name “_namemap”.

```
class bond_command : public command_i
{
    public:
```

```
        bond_command( const string& atom1, const string& atom2, int order );
        virtual ~bond_command( );
        virtual bool exec( );

    private:
        string m_atom1;
        string m_atom2;
        int m_order;
};
```

Class bond_command: Add one bond between two atoms.

```
class bondbydis_command : public command_i
{
    public:
        bondbydis_command( const string& object, double cutoff );
        virtual ~bondbydis_command( );
        virtual bool exec( );

    private:
        string m_object;
        double m_cutoff;
};
```

Class bondbydis _command: Determine bonds according to the cutoff. “object” is molecule name stored in **content()**.

```
class center_command : public command_i
{
    public:
```

```
        center_command( const string& object );
        virtual ~center_command( );
        virtual bool exec( );

    private:
        string m_object;
};
```

Class center_command: Print the geometric center's coordinate of the molecule which is stored in **content()** named "object".

```
class charge_command : public command_i
{
    public:
        charge_command( const string& object );
        virtual ~charge_command( );
        virtual bool exec( );

    private:
        string m_object;
};
```

Class charge_command: Print the charge of the molecule which is stored in **content()** named "object".

```
class check_command : public command_i
{
    public:
        check_command( bool checkmolcharge, bool checkatomparams,
                       bool checkbondparams, bool checkangleparams,
```

```
                    bool checkbondlength,  bool checkatomclashes,
                    const string&  molnam  );
        virtual  ~check_command();
        virtual  bool exec();

    private:
        bool m_checkmolcharge;
        bool m_checkatomparams;
        bool m_checkbondparams;
        bool m_checkangleparams;
        bool m_checkbondlength;
        bool m_checkatomclashes;
        string  m_molnam;
};
```

Class check_command: Check the parameters set in molecule "molnam". Parameters are used to turn on/off the corresponding check.

```
class copy_command  : public  command_i
{
    public:
        copy_command(  const string&  dst, const string&  src );
        virtual  ~copy_command(  );
        virtual  bool exec( );

    private:
        string  m_src;
        string  m_dst;
};
```

Class copy_command: Copy the molecule "src", and then put it into **content()** with

name “dst”.

```
class create_command : public command_i
{
    public:
        create_command( const string& object, const string& dest, const string& name );
        create_command( const string& object, const string& dest, const string& name, const string& type, const string& charge );
        virtual ~create_command();
        virtual bool exec();

    private:
        string m_object;
        string m_dest;
        string m_name;
        string m_type;
        string m_chrg;
};
```

Class create_command: Function with 5 arguments can create atom (object:atom) named “name” with “type” and “charge”. Function with 3 arguments can create residue/unit (object:residue/unit) named “name”, all the objects created is stored in molecule “dest”, and then be set into **content()**.

```
class delete_command : public command_i
{
    public:
        delete_command( const string& action, const string& oper1 );
```

```
        delete_command( const string& action, const string& oper1, const string&
oper2 );
        virtual ~delete_command( );
        virtual bool exec( );

    private:
        string m_action;
        string m_oper1;
        string m_oper2;
};
```

Class delete_command: Function with 2 arguments can delete the atom/residue (action: deleteatom/deletebond) stored in **content()** named "oper1". Function with 3 arguments can delete bonds (action: deletebond) stored in **content()** with two atoms named "oper1" and "oper2".

```
class energy_command : public command_i
{
    public:
        energy_command(const string& name, const string& parm);
        energy_command(const string& name, const string& parm, const string&
vec1, const string& vec2);
        virtual ~energy_command();
        virtual bool exec();

    private:
        string m_name;
       string m_parm;
        string m_vec1;
        string m_vec2;
```

};

Class energy_command: Calculate the energy of molecule “name” stored in **content()** according to the “parm” set. “parm” is a string composed of igb and cut, such as “igb,1:cut,12”, “ ,:” can be used as separators. Energy is divided into two parts: bond energy and non-bond energy. Bond energy contains the energy of Bond, Angle and Torsion. Non-bond energy can be calculated by using different means. If “igb=0”, it will use Ewald summation to calculate the energy, it adopts periodic boundary condition. If “igb!=0”, energy of Vdw and Electrostatic will be calculated first, and a GB model will be used. Function with “vec1” and “vec2” will only calculate the non-bond energy between part “vec1” and “vec2” of molecule “name” which is stored in **content()** by using GB model. Function without “vec1” will calculate the total energy.

```
class impose_command  : public  command_i
{
    public:
        impose_command( const string& unit, const vector< string >& units, const vector< vector< string  > >& inters  );
        virtual  ~impose_command(  );
        virtual  bool exec( );

    private:
        string  m_unit;
        vector< string  > m_resds;
        vector< vector< string  > > m_inters;
};
```

Class impose_command: Impose the interaction “inters” to the residue “units” of molecule “unit”.

```
class loadcrd_command : public command_i
{
    public:
        loadcrd_command( const string& action, const string& name, const string&
file );
        virtual ~loadcrd_command();
        virtual bool exec();

    private:
        string m_action;
        string m_name;
        string m_file;
};
```

Class loadcrd_command: Load the coordinate's file of amoeba force field into molecule "name". action should be "loadamoebacrd".

```
class loadfrc_command : public command_i
{
    public:
        loadfrc_command( const string& action, const string& file );
        virtual ~loadfrc_command();
        virtual bool exec();

    private:
        string m_file;
        string m_action;
};
```

Class loadfrd_command: Load parameters from force field files. "action" has two

types: “loadamberparams” and “loadamoebaparams”. Parameters are stored in “_amberffp” of **content()**.

```
class loadmol2_command : public command_i
{
    public:
        loadmol2_command( const string& name, const string& file );
        virtual ~loadmol2_command();
        virtual bool exec();

    private:
        string m_name;
        string m_file;
};
```

Class loadmol2_command: Load the molecule from MOL2 formatted file.

```
class loadoff_command : public command_i
{
    public:
        loadoff_command( const string& file, const string& name );
        virtual ~loadoff_command( );
        virtual bool exec( );

    private:
        string m_file;
        string m_name;
        vector< string > m_names;
};
```

Class loadoff_command: Load the parameters from lib file which is stored in dat/lib, usually invoked in files located in /dat/cmd, such as “leaprc.ff99”.

```
class loadpdb_command : public command_i
{
    public:
        loadpdb_command( const string& action, const string& name, const string& file, const string& seq );
        virtual ~loadpdb_command();
        virtual bool exec();

    private:
        string m_action;
        string m_name;
        string m_file;
        string m_seq;
};
```

Class loadpdb_command: Load the molecule from PDB formatted file or sequence. “action” can be “loadpdb” or “loadpdbusingseq”. “file” is the file name specified for “loadpdb” and “seq” for “loadpdbusingseq”.

```
class loadprep_command : public command_i
{
    public:
        loadprep_command(const string& file, const string& prefix);
        virtual ~loadprep_command();
        virtual bool exec();
```

```
    private:
        string m_file;
        string m_prefix;
        vector< string > m_names;
};
```

Class loadprep_command: Load each residue's descriptions from the force field files. They are stored in /dat/prep directory, and have a ".in" extension. "prefix" is the name set in **content()**.

```
class loadsdf_command : public command_i
{
    public:
        loadsdf_command( );
        loadsdf_command( const string& name, const string& file );
        virtual ~loadsdf_command();
        virtual bool exec();

    private:
        string m_name;
        string m_file;
};
```

Class loadsdf_command: Load the molecule from SDF formatted file.

```
class measure_command : public command_i
{
    public:
        measure_command( const vector< string >& atoms );
        virtual ~measure_command( );
```

```
        virtual bool exec( );

    private:
        vector< string > m_atoms;
};
```

Class measure_command: Print the distance, angle or torsion angle if "atoms" contains 2, 3 or 4 atoms.

```
class merge_command : public command_i
{
    public:
        merge_command( const string& action, const string& name, const string&
list );
        virtual ~merge_command( );
        virtual bool exec( );

    private:
        string m_action;
        string m_name;
        string m_list;
};
```

Class merge_command: read the residues from "list", the format can be like {ALA LEU ARG…}, the action should be sequence, we will add more actions in the future, and the residues are stored in **content()** with the name "name".

```
class moloper_command : public command_i
{
    public:
```

```
        moloper_command( const string& action, const string& mname );
        virtual ~moloper_command();
        virtual bool exec();

    private:
        string m_action;
        molecule_ptr m_pmol;
};
```

Class moloper_command: Action can be “fixbond”, “addhydr”, “setpchg” and “parmchk”, which is used to do some preparations for the molecule before do other operations. The meaning of each action can be seen in the supporting information. “mname” is the molecule name stored in **content()**.

```
class savemol2_command : public command_i
{
    public:
        savemol2_command( );
        savemol2_command( const string& name, const string& file );
        virtual ~savemol2_command();
        virtual bool exec();

    private:
        string m_name;
        string m_file;
};
```

Class savemol2_command: Save the molecule in the MOL2 format file.

```
class saveoff_command : public command_i
```

```
{
    public:
        saveoff_command( const string& unit, const string& file );
        virtual ~saveoff_command( );
        virtual bool exec( );

    private:
        string m_unit;
        string m_file;
};
```

Class saveoff_command: Save the residue's information into OFF format files.

```
class savepdb_command : public command_i
{
    public:
        savepdb_command( const string& name, const string& file );
        virtual ~savepdb_command();
        virtual bool exec();

    private:
        string m_name;
        string m_file;
};
```

Class savepdb_command: Save the molecule "name" into PDB format file.

```
class saveprm_command : public command_i
{
    public:
```

```
        saveprm_command( const string& action, const string& name, const string& top, const string& xyz );
        virtual ~saveprm_command();
        virtual bool exec( );

    private:
        string m_name;
        string m_action;
        string m_topfile;
        string m_xyzfile;
};
```

Class saveprm_command: Save the parameters of molecule "name" into two separate files, which can be the input files for amber's simulation. "top" is the topology file's name, and xyz the coordinate file. Action can be "savetinkerparm", "saveamberparm", "saveamberparmpol" and "saveamoebaparm", their meanings can be seen in Supporting Information.

```
class savesdf_command : public command_i
{
    public:
        savesdf_command( const string& name, const string& file );
        virtual ~savesdf_command();
        virtual bool exec();

    private:
        string m_name;
        string m_file;
};
```

Class savesdf_command: Save the molecule "name" into SDF format file.

```
class solvate_command : public command_i
{
    public:
        solvate_command( const string& action, const string& solute, const string& solvent, const string& center, const string& extent, double closeness, bool iso );
        virtual ~solvate_command( );
        virtual bool exec( );

    private:
        string m_action;
        string m_solute;
        string m_center;
        string m_solvent;
        string m_extent;
        double m_closeness;
        bool m_iso;
};
```

Class solvate_command: Add the solvent "solvent" (usually HOH) to the solute "solute". Action can be "solvatebox", "solvateoct", "solvatecap" and "solvateshell", they represent different kinds of solvation models. Parameter set can be seen in Supporting Information.

```
class transform_command : public command_i
{
    public:
        transform_command( const string& action, const string& object, const string& offset );
```

```
        virtual ~transform_command( );
        virtual bool exec( );

    private:
        string m_action;
        string m_object;
        string m_offset;
};
```

Class transform_command: Transform the object “object” by using the given “offset”. Action can be “rotate” and “translate”, the format of “offset” has been listed in their corresponding function of Supporting Information.

```
class zmatrix_command : public command_i
{
    public:
        zmatrix_command( const string& object, const vector< vector< string > >&
inters );
        virtual ~zmatrix_command( );
        virtual bool exec( );

    private:
        string m_object;
        vector< vector< string > > m_inters;
};
```

Class zmatrix_command: Transform the internal coordinates “inters” of the molecule or residue “object” into Cartesian coordinates.