Object-Oriented Programming in Simulations

Jeongnim Kim
Department of Physics, Ohio State University

Object-Oriented Programming in Simulations

- Introduction to Object-Oriented programming
  - Procedural programming: algorithms
  - Modular programming: data abstraction
  - Object-Oriented programming: data abstraction and interfaces
  - Basics of Objected-Oriented programming
- Practical implementation of OO programming in C++
  - Communicating OO design
  - Design patterns : reuse of algorithms
  - Templates : generic programming
  - Parallel programming and Object-Oriented frameworks

Why Object-oriented frameworks?

- Growing roles of simulations:
  - Atomistic simulations using inter-atomic potentials:

    Classical – Empirical Tight-binding – *Ab initio* potentials
  - Electronic structures of nanoscale heterostructures.
  - Macroscopic modeling tools based on microscopic information.
  - Algorithm developments in materials simulations.
- Ever-changing computing environments: RISC workstations, Intel workstations, Distributed Memory Processing (DMP), Shared Memory Processing (SMP) and SMP/DMP hybrids.
- Evolving research teams and research projects.

Object-oriented frameworks facilitate development of flexible, easy-to-use and efficient programs.

Procedural programming: Algorithms

- Procedural programming: functional decomposition.
  - Execute a sequence of algorithms to solve a problem.
  - Perform specific tasks (functions) on data structures.
  - Provide unlimited access to the data structures.

- Some drawbacks of Procedural programming
  - Difficult to localize data shared by different parts of a program.
    - Moving up of common data → everything in global name space?
    - Communication between functions are mediated by a long list of arguments or common blocks /global variables → error prone.
  - Difficult to reuse existing codes and to develop large scale programs incrementally.
  - Difficult to port the codes to new computing platforms.

Procedural programming: MD code

aMD.cpp

```
main() {
   const int nat = 10;
   double R[nat][3];
   double V[nat][3];
   double F[nat][3];


   int mdstep = 100;

   initialize(R,V);

   int step = 0;
   while(step<mdstep){

    getForce(R, F);
    update(R, F, V);
    report(R);
    step++;

   }
}
```

mdFunctions.cpp

```
void
initialize(double R[][3], double V[][3]){

  assign position;
  assign velocity;

}

void
getForce(double R[][3], double F[][3]){

  // a Model potential
  assign Forces;

}

void
update(double R[][3], double F[][3],
       double V[][3]) {

 loop over:
  R[i][j] += c1*V[i][j]+c2*F[i][j];

}
```

Compile to get **aMD.o** and **mdFunctions.o** and link them.

Procedural programming: aMD code + change

aMD.cpp

```
main() {
   const int nat = 10;
   double R[nat][3];
   double V[nat][3];
   double F[nat][3];
   double R1[nat][3];

   int mdstep = 100;

   initialize(R, R1, V);

   int step = 0;
   while(step<mdstep){

    getForce(R, F);
    update(R, R1, F, V);
    report(R);
    step++;

   }
}
```

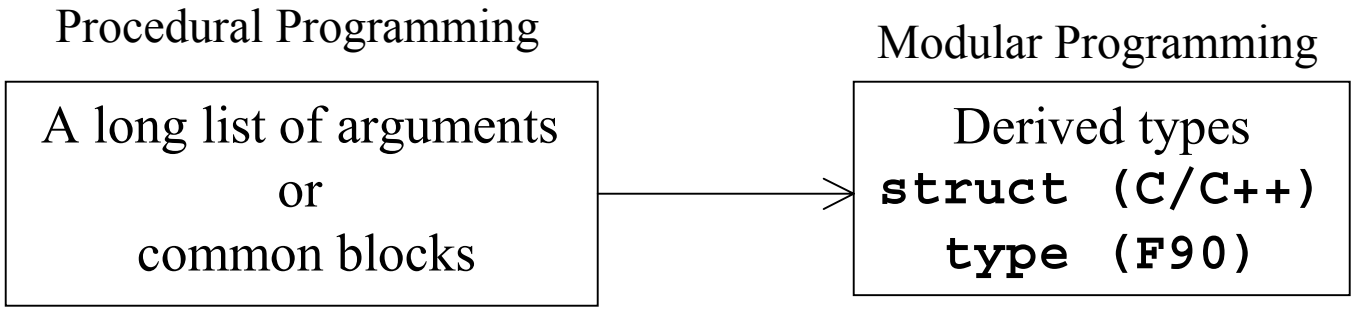
mdFunctions2.cpp

```
void
initialize(double R[][3], double R1[][3],
           double V[][3]){

  assign position R;
  assign previous position R1;
  assign velocity;
}

void
getForce(double R[][3], double F[][3]){

  // a Model potential
  assign Forces;
}

void
update(double R[][3], double R1[][3],
       double F[][3], double V[][3]) {

 loop over:
  R[i][j] =
   c1*V[i][j]+c2*F[i][j]+c3*R1[i][j];
}
```

Compile to get **`aMD.o`** and **`mdFunctions2.o`** and link them.

Modular programming: Data abstraction

- Define a user-derived type which contains related data.



- Provide external functions that operate on the user-derived types and built-in types.
- Example of a user-defined type: 3-D vector

C/C++

```
struct Vec3D {
  double data[3];
};
```

F90

```
type:: Vec3D
  real, dimension:: data(3)
end type Vec3D
```

Procedural programming vs. Modular programming

aMD.cpp

```
main() {
  const int nat = 10;
  double R[nat][3];
  double V[nat][3];
  double F[nat][3];

  int mdstep = 100;

  initialize(R, V);

  int step = 0;
  while(step<mdstep){

   getForce(R, F);
   update(R, F, V);
   report(R);
   step++;

  }
}
```

modularMD.cpp

```
#include "PtclSet.h"
main() {

 PtclSet mySys;

 initialize(&mySys);
  int step = 0,mdstep = 100;
  while(step<mdstep){
   getForce(&mySys);
   update(&mySys);
   report(&mySys);
   step++;
  }
}
```

PtclSet.h

```
struct Vec3D {double data[3];};

struct
PtclSet{Vec3D *R, *V, *F;};

void initialize(PtclSet*);
void getForce(PtclSet*);
void update(PtclSet*);
```

encapsulated

Modular programming: Data abstraction

- Improve data localization in modules: user-defined types contain a collection of data associated with the types, and modules contain related functions.
- User-defined types as well as built-in types can be used as arguments of external functions.

- Distribution of data and functionality of each module require much care at the design stage.
- Implementations of modules are exposed to the users → Changes in a user-defined type have a direct impact on the modules that use the type.
- Module inter-dependency can increase compile time → Beware of circular dependency.
- Memory management is done by external functions → Demand disciplined use of dynamic memory allocation.

Object-Oriented programming

- What are Object-Oriented user-defined types (e.g., classes in C++)?
  - represent physical concepts.
  - encapsulate data: Modular programming.
  - provide procedures (member functions and operators) on its data.
- Ideal classes have minimum dependence on outside world and provides interfaces that expose a minimal amount of information.

- A problem is solved by interactions between classes via class interfaces and external functions.

- Class design for ***reusable*** software involves decisions on
  - Distribution of data and functionality among classes representing the physical concepts.
  - Kinds of relationships between interacting classes.

Modular programming vs. OO programming

modularMD.cpp

```
#include "PtclSet.h"
main() {
  PtclSet mySys;

  initialize(&mySys);
  int step = 0, n= 100;
  while(step<n){
   getForce(&mySys);
   update(&mySys);
   report(&mySys);
   step++;
  }
}
```

create an object

ooMD.cpp

```
#include "ooPtclSet.h"
main() {
  PtclSet mySys;

  mySys.initialize();
  int step = 0, n = 100;
  while(step<n){
   mySys.getForce();
   mySys.update();
   mySys.report();
   step++;
  }
}
```

PtclSet.h

```
struct PtclSet{
  Vec3D *R, *V, *F;
};

void initialize(PtclSet*);
void getForce(PtclSet*);
void update(PtclSet*);
```

ooPtclSet.h

```
class PtclSet{
prviate:
  Vec3D *R, *V, *F;
public:
  PtclSet();
  ~PtclSet();
  void initialize();
  void getForce();
  void update();
};
```

OO programming: Class relationships

- Classes are basic entities in OO programming.
- Software development of OO frameworks considers
  - Distribution of data and functionality among classes representing the physical concepts
  - Kinds of relationships between interacting classes

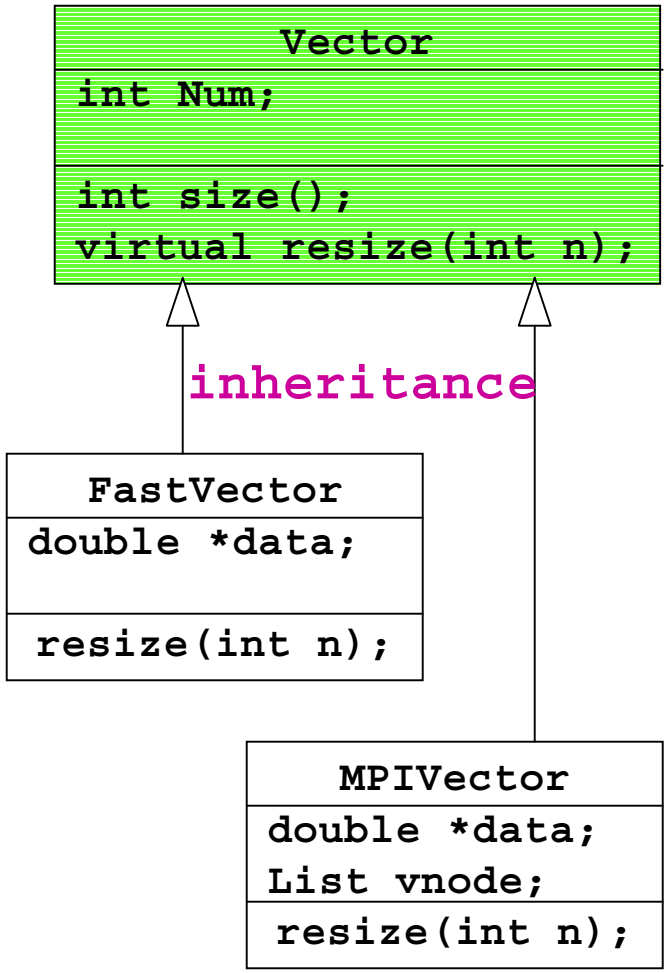
  to maximize reuse of the classes and to hide class information from outside world.

- Class relationships
  - IsA : inheritance or parameterization (template) of classes.
  - Uses-in-the-interface : interactions through class interfaces.
  - Uses-in-the-implementation : usage as data members or local variables.

OO programming : IsA relationship

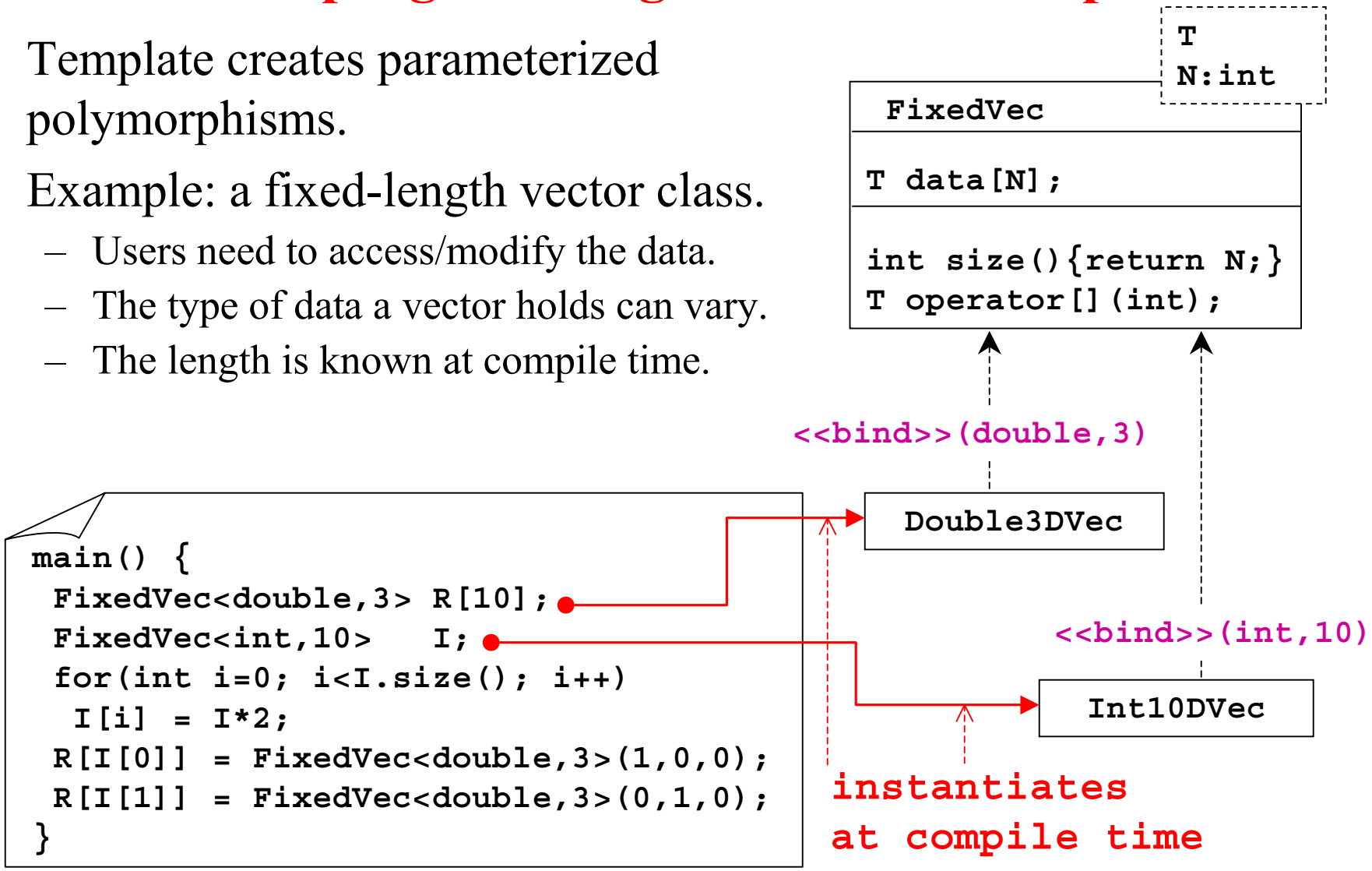
- Inheritance creates class hierarchy.
- Derived classes inherit data members and interfaces of a parent class

  → reuse of common interfaces.
- Derived classes typically overwrite virtual functions of its parent class and provide its implementations.
- Example: vector class.
  - A vector contains a set of data.
  - Users need to access/modify the data.
  - How the data is stored can vary.



OO programming : IsA relationship

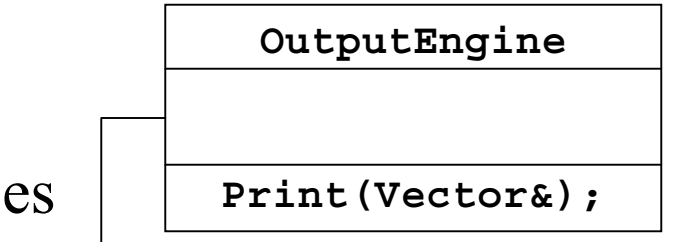
- Template creates parameterized polymorphisms.
- Example: a fixed-length vector class.
  - Users need to access/modify the data.
  - The type of data a vector holds can vary.
  - The length is known at compile time.



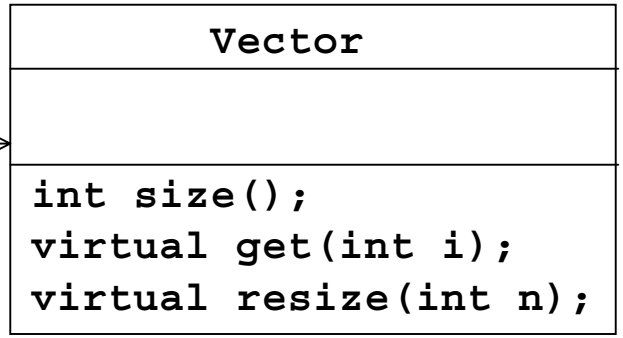
OO programming: Used-in-the-interface relationship

- A class uses other classes or its own class as arguments.
- The user class knows public interfaces of the classes being used.

- Example: OuputEngine for a Vector.
  - Will use a specialized file format to print a Vector.
  - Knows how to access an element of a Vector.
  - Does not care how the data of a Vector is stored.



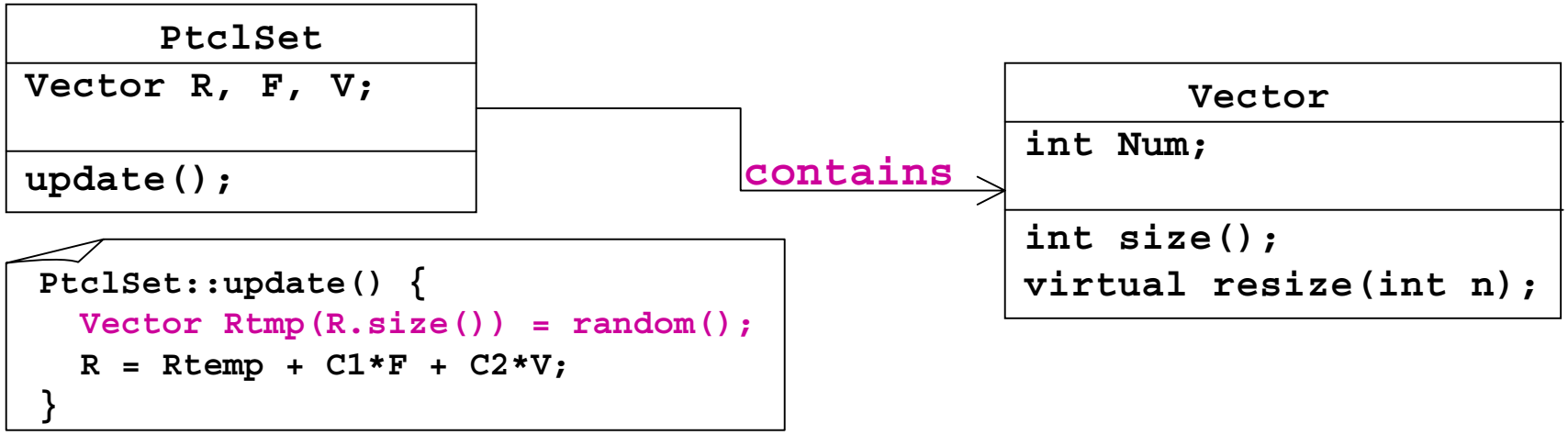
uses




```
OuputEngine::Print(Vector& a) {
 for(int i=0; i<a.size(); i++) {
  cout << i << " " << a.get(i)<< endl;
}
```

OO programming: Used-in-the-implementation

- A class uses other classes as data members or local variables in the implementation of its interfaces.
- The user class knows public interfaces of the class being used.
- Example: A Particle class for a set of particles.
  - Particle attributes (position, velocity …) are stored in Vectors.
  - A PtclSet object contains three Vectors (**`R,F,V`**) and uses a temporary Vector object to perform **`PtclSet::update().`**



Modular programming vs. OO programming

modularMD.cpp

```
#include "PtclSet.h"
main() {
  PtclSet mySys;

  initialize(&mySys);
  int step = 0, n= 100;
  while(step<n){
   getForce(&mySys);
   update(&mySys);
   report(&mySys);
   step++;
  }
}
```

create an object

ooMD.cpp

```
#include "ooPtclSet.h"
main() {
  PtclSet mySys;

  mySys.initialize();
  int step = 0, n = 100;
  while(step<n){
   mySys.getForce();
   mySys.update();
   mySys.report();
   step++;
  }
}
```

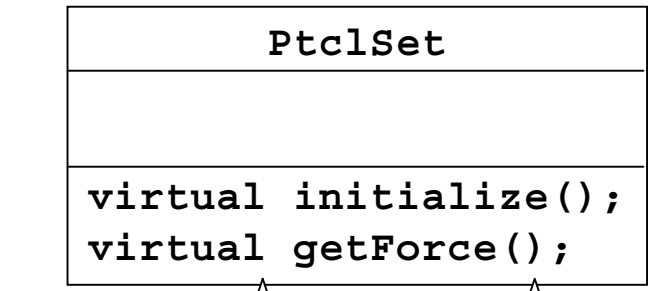
PtclSet.h

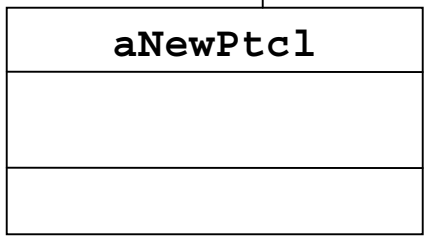
```
struct PtclSet{
  Vec3D *R, *V, *F;
};

void initialize(PtclSet*);
void getForce(PtclSet*);
void update(PtclSet*);
```

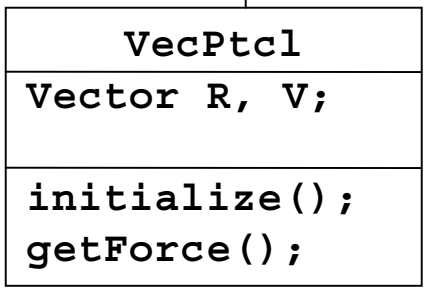
ooPtclSet.h

```
class PtclSet{
prviate:
  Vec3D *R, *V, *F;
public:
  PtclSet();
  ~PtclSet();
  void initialize();
  void getForce();
  void update();
};
```

OO programming : ooMD code







ooPtclSet.h

```
class PtclSet{
public:
 PtclSet();
 ~PtclSet();
 virtual void initialize() =0;
 virtual void getForce() =0;
 virtual void update() =0;
};

PtclSet* create(int i);
```

Declare interfaces
No implementations

VecPtcl.h

```
#include "ooPtclSet.h"

class VecPtcl: public PtclSet{
public:
 VecPtcl();
 ~VecPtcl();
 void initialize();
 void getForce();
 void update();
};
```

OO Programming : ooMD code

ooPtclSet.h

```
class PtclSet{
public:
 PtclSet();
 ~PtclSet();
 virtual void initialize() =0;
 virtual void getForce() =0;
 virtual void update() =0;
};

PtclSet* create(int i);
```

ooMD.cpp

```
#include "ooPtclSet.h"
main() {
 PtclSet* mySys = create(0);

 mySys->initialize();
 int step = 0, n = 100;
 while(step<n){
  mySys->getForce();
  mySys->update();
  mySys->report();
  step++;
 }
}
```

Link

VecPtcl.h

```
#include "ooPtclSet.h"

class VecPtcl: public PtclSet{
public:
 VecPtcl();
 ~VecPtcl();
 void initialize();
 void getForce();
 void update();
};
```

VecPtcl.cpp

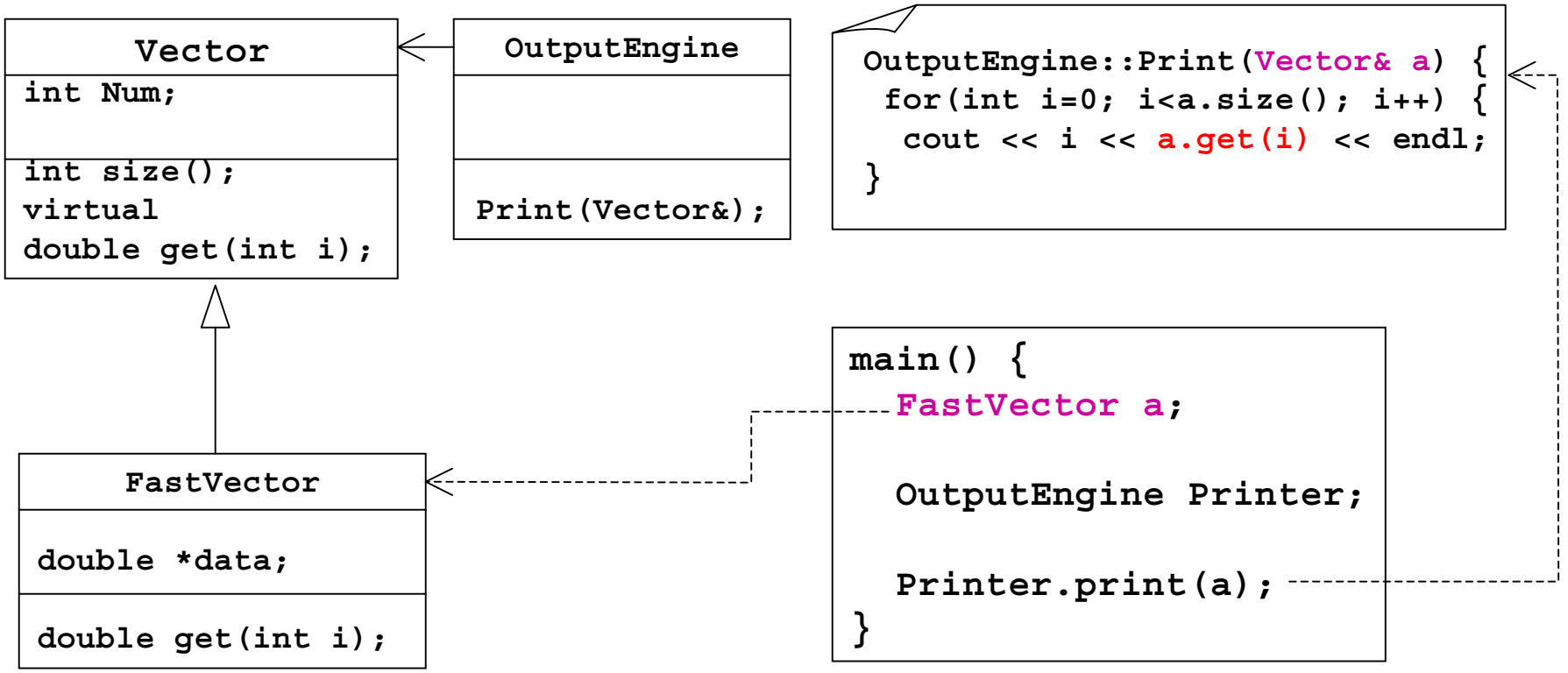
```
#include "VecPtcl.h"

Implement
VecPtcl::initialize();
VecPtcl::getForce();
VecPtcl::update();

PtclSet* create(int i) {
  if(i != 0)
  cout << "Bad option ignored.\n";
  return new VecPtcl;
}
```

Object-Oriented programming

1. A class encapsulates data and provides procedures (member functions and operators) on its data.
2. Inheritance creates class hierarchy and enables reuse of common interfaces provided by parent classes.
3. Virtual functions enable run-time binding.
4. Abstract virtual functions enable decoupling of implementations from interfaces: reduce the dependency between classes and facilitates separate compilations of concrete classes.
5. Parameterized polymorphism using template features facilitates implementation of generic algorithms (e.g., STL).

OO programming : Inheritance

2. Inheritance creates class hierarchy and enables reuse of common interfaces provided by parent classes.
3. Virtual functions enable run-time binding.



Decoupling implementations from interfaces

ooPtclSet.h

```
class PtclSet{
public:
 PtclSet();
 ~PtclSet();
 virtual void initialize() =0;
 virtual void getForce()=0;
 virtual void update() =0;
};

PtclSet* create(int i);
```

ooMD.cpp

```
#include "ooPtclSet.h"
main() {
 PtclSet* mySys = create(0);

 mySys->initialize();
 int step = 0, n = 100;
 while(step<n){
  mySys->getForce();
  mySys->update();
  mySys->report();
  step++;
 }
}
```

Link

VecPtcl.h

```
#include "ooPtclSet.h"

class VecPtcl: public PtclSet{
public:
 VecPtcl();
 ~VecPtcl();
 void initialize();
 void getForce();
 void update();
};
```

VecPtcl.cpp

```
#include "VecPtcl.h"

Implement
VecPtcl::initialize();
VecPtcl::getForce();
VecPtcl::update();

PtclSet* create(int i) {
  if(i != 0)
  cout << "Bad option ignored.\n";
  return new VecPtcl;
}
```

Generic programming

FixedVector.h

```
template<class T, unsigned N>
class Vector{
 T data[N];
public:
 Vector();
 ~Vector();

 inline T operator[](int i)
 { return data[i];}
 Vector& operator*=(T x);

       complete operations
};
```

myapp.cpp

```
#include <vector> //STL vector
#include "FixedVector.h"
main(){
 typedef
 Vector<double,3> Pos_t;

 vector<Pos_t> A(5), B(5);
 vector<Pos_t> C(5);
 C = A + 2.0*B;
}
```

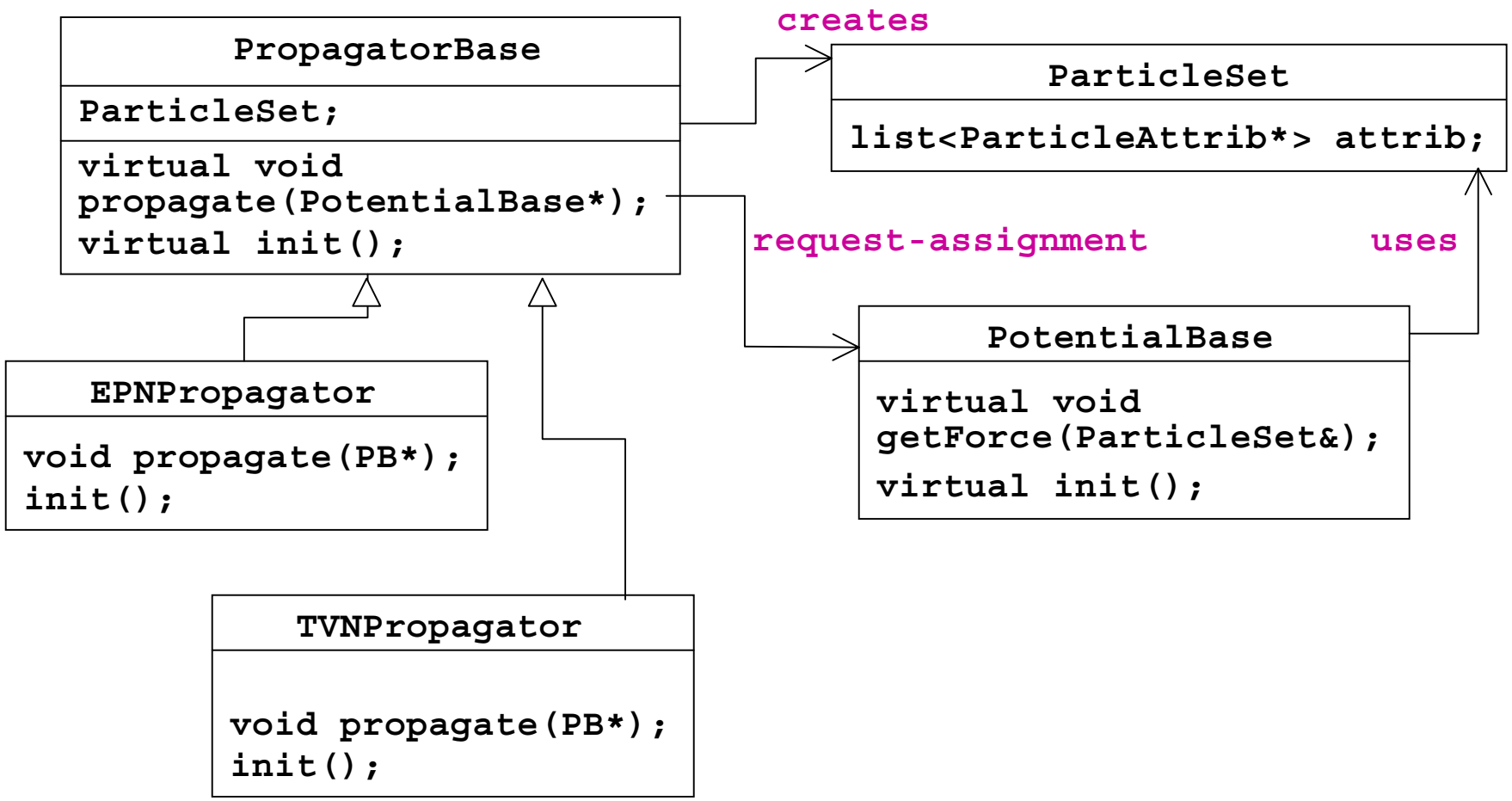
- Parameterization of containers, iterators and algorithms, e.g., Standard Template Library (STL).
- Concrete objects are instantiated at compile time.
- Optimization is achieved by using inline functions and operator overwriting.
- Special cases can be hand-written for optimization on specific architectures.

Communicating Object-Oriented design

- Object-Oriented design involves decisions on:
  - Decomposition of a problem.
  - Representation of physical concepts as classes.
  - Relationships between classes.
  - Class interfaces: member functions and operators.
  - Concrete implementations of interfaces.

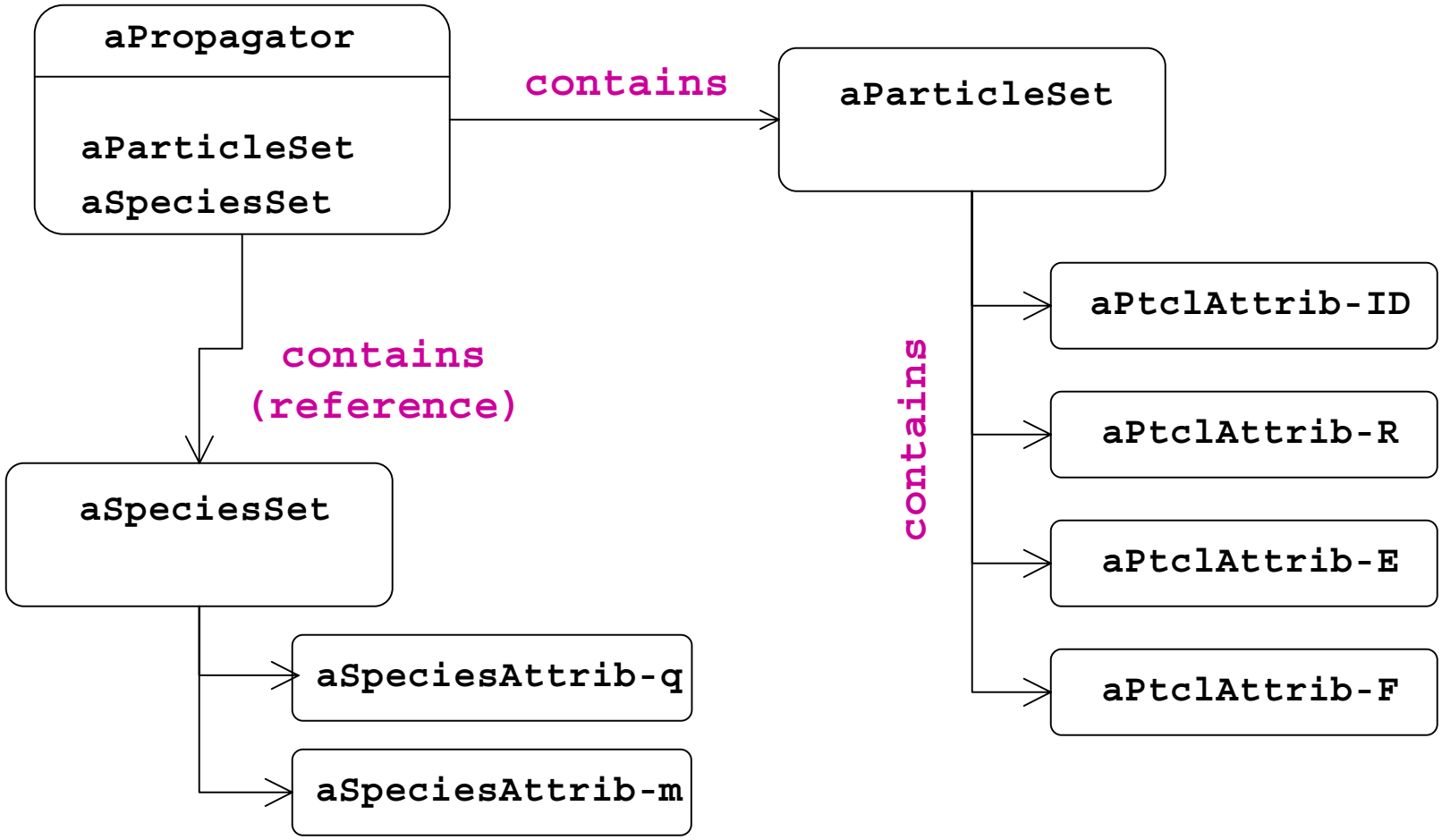
- Unified Modeling Language (UML) assists OO design and analysis.
  - Class diagrams: classes, interfaces and class relationships.
  - Object diagrams: a particular object structure at run-time.
  - Interaction diagrams: flow of requests between classes.

Class diagram: classes, interfaces and relationships

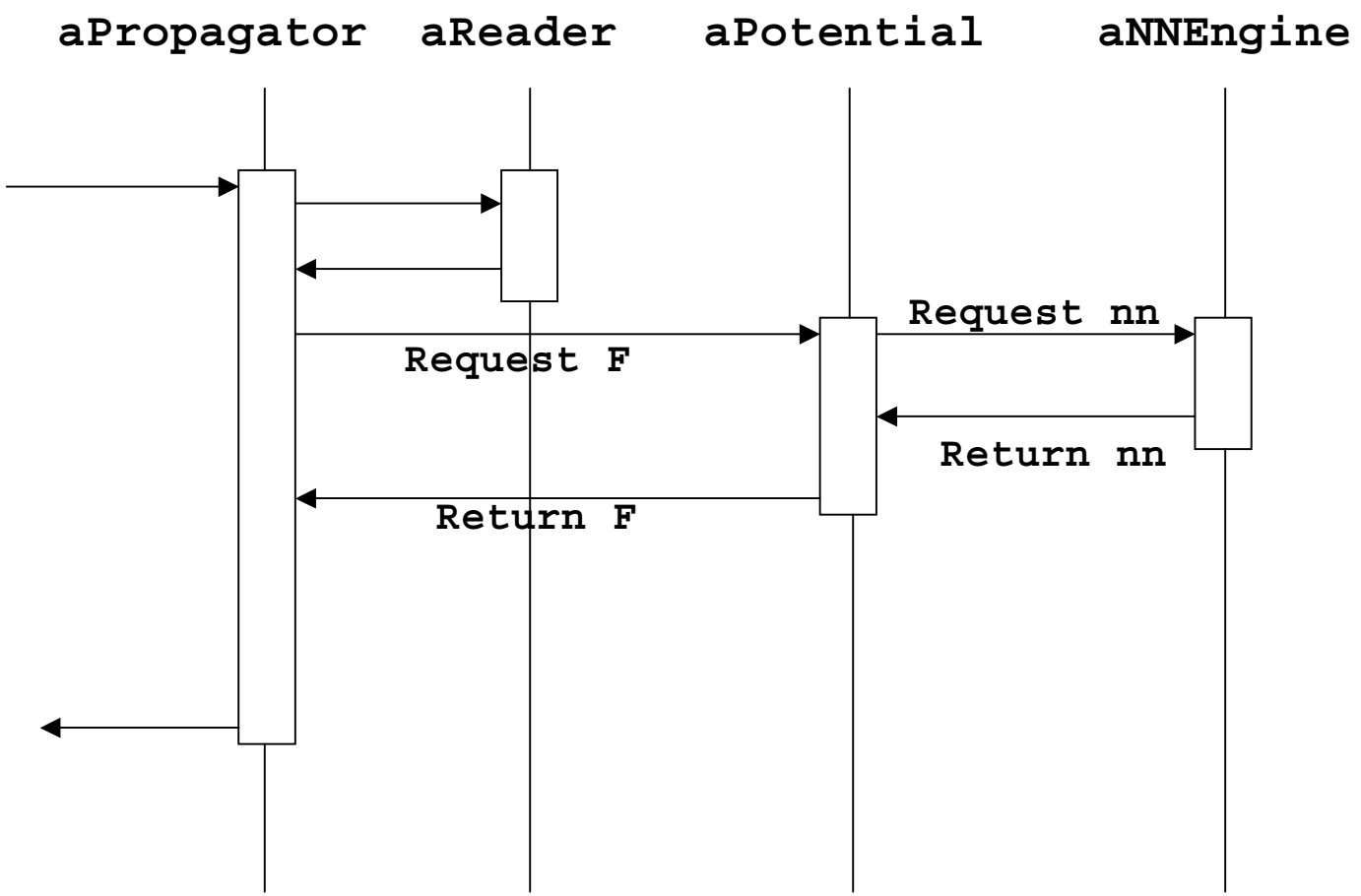


PB = PotentialBase

Object diagram: a particular object structure at run-time



Interaction diagram: flow of requests between classes



Design patterns

- What are **Patterns**: abstractions from concrete forms which keep recurring in different contexts.*
- Patterns capture
  - static and dynamic structures, and
  - relationships and interactions between components

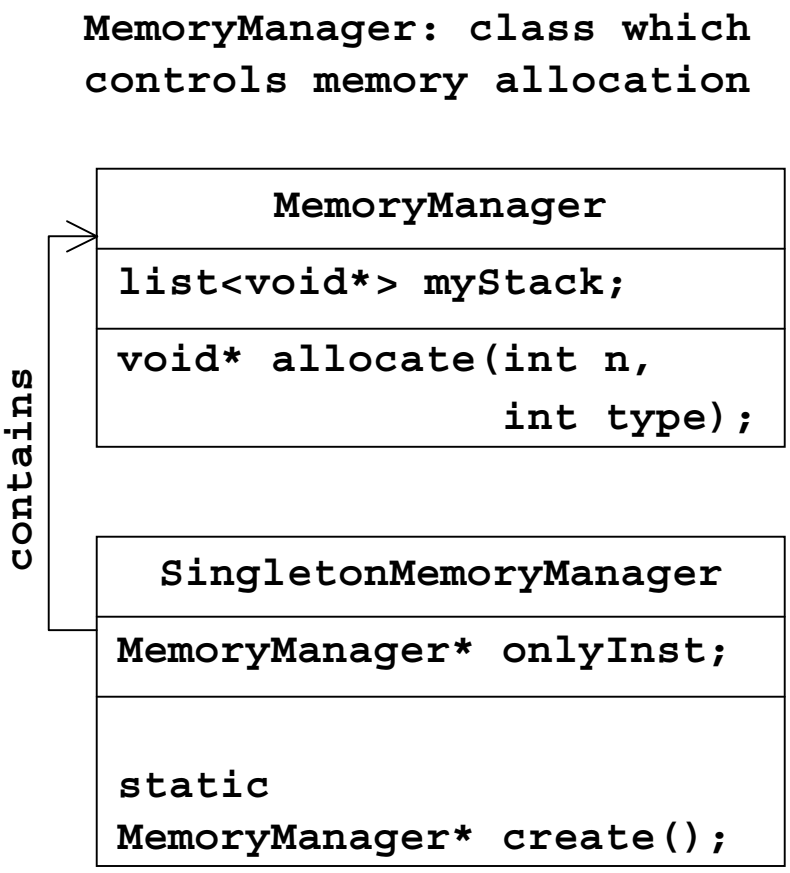
  of successful solutions to the recurring problems.
- Patterns provide ***reusable*** solutions to specific problems.
- Short list of patterns
  - Singleton pattern
  - Builder pattern
  - Composite pattern
  - Iterator pattern

*Design Patterns: elements of reusable Object-Oriented Software, Gamma *et al* (95).

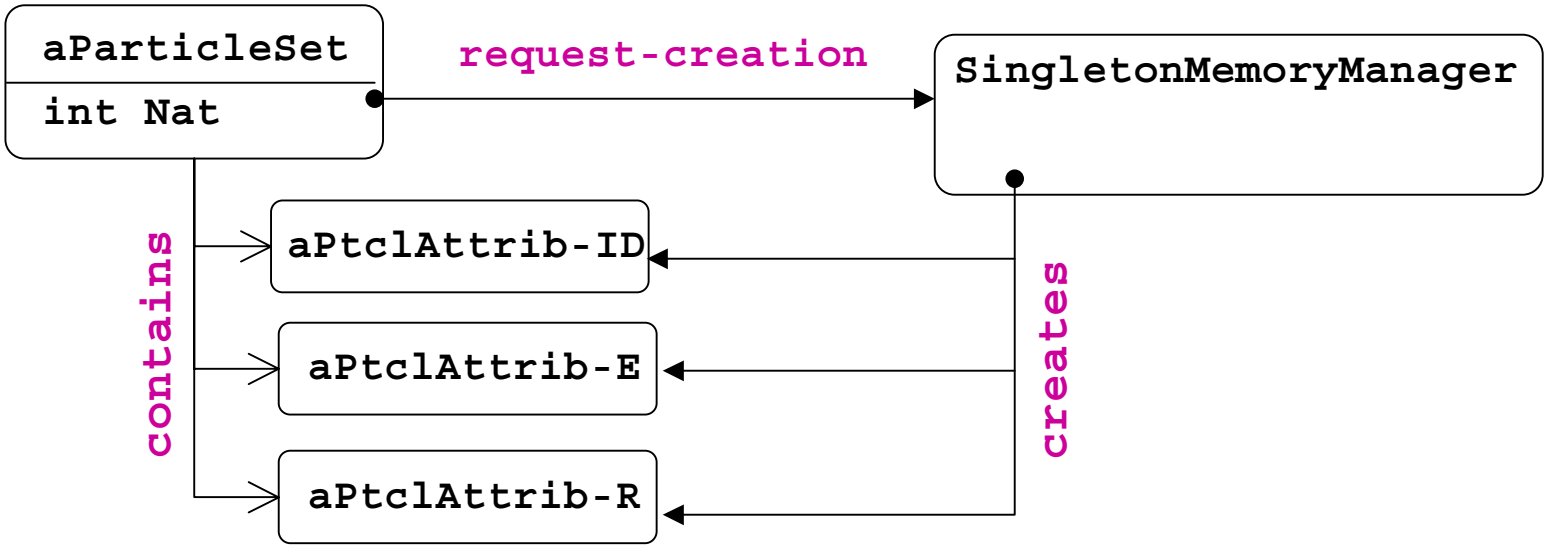
Singleton pattern: a design pattern

- The singleton pattern ensures that only one instance of a class is created.

- Problems: a resource manager or a large object shared by many objects
- Old solution: declare it as a global variable.
- Solution by the singleton pattern: Create a class which instantiate the object once and returns a reference to the unique instance.



Object diagram of the singleton pattern

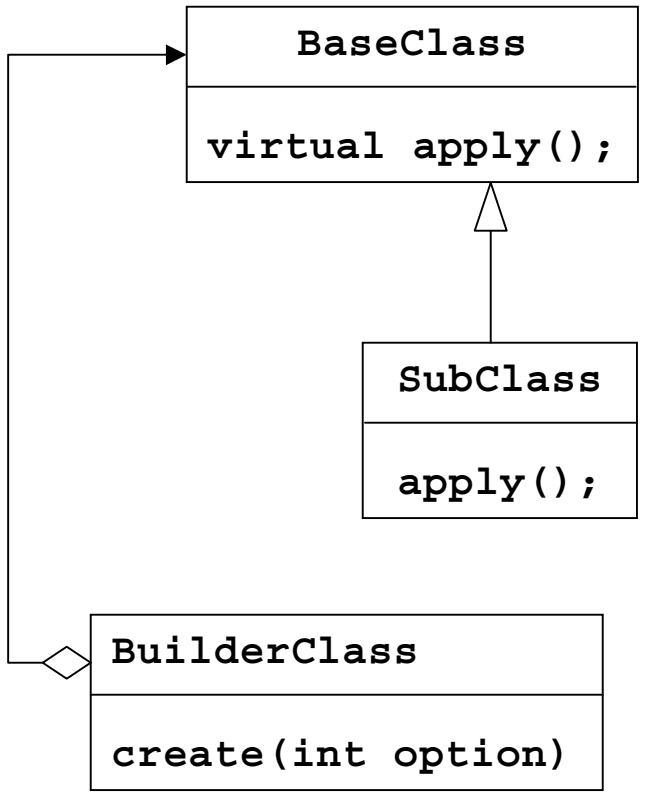



```
PtclSet::addAttrib() {

  MemoryManager* manager = SingletonMemoryManager::create();
  ID = manager->allocate(Nat,INTEGER);// integer type
  E = manager->allocate(Nat,SCALAR);  // scalar type
  R = manager->allocate(Nat,VECTOR);  // vector type

}
```

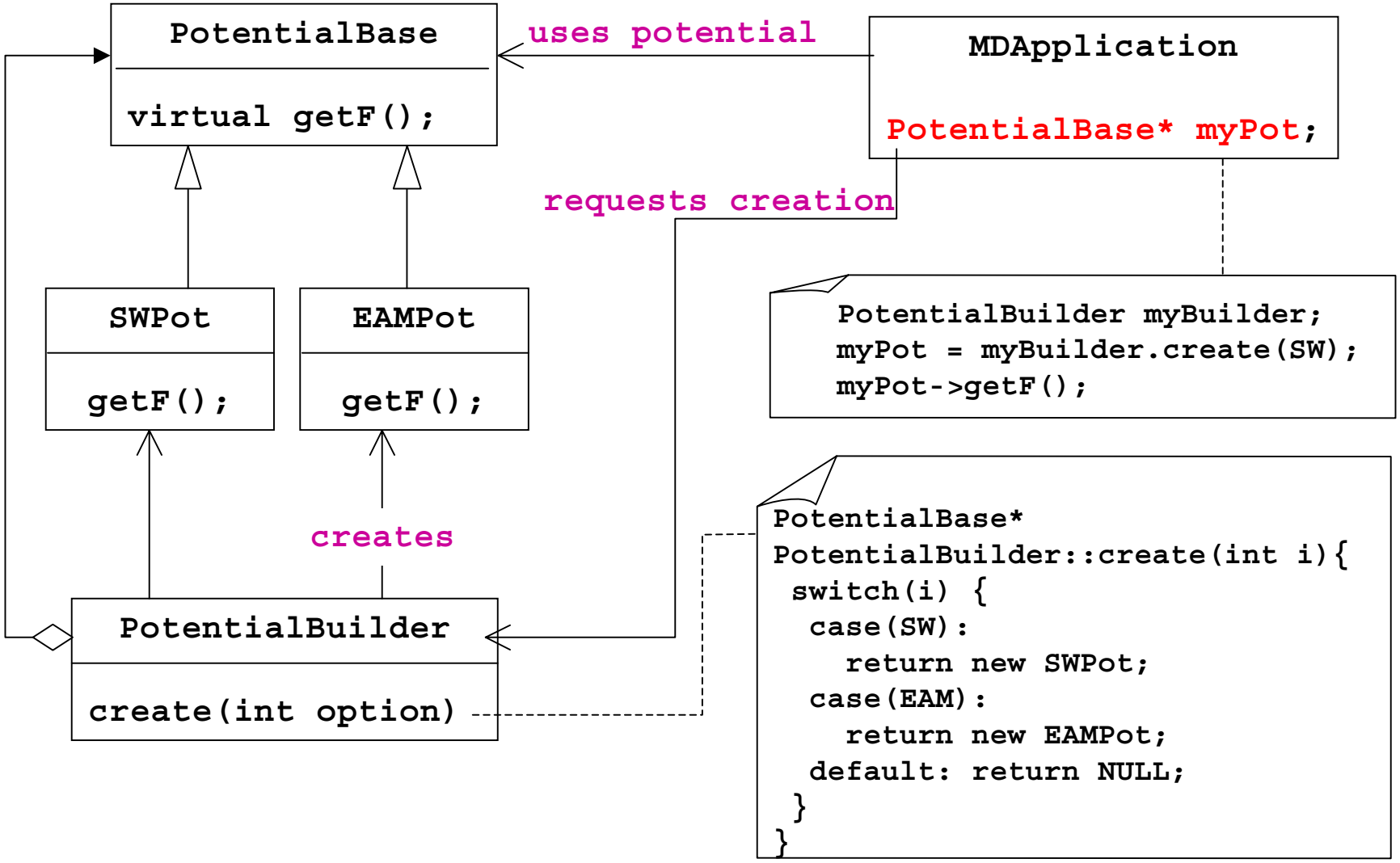
Builder pattern: a design pattern

- The builder pattern separates the construction of a complex object from its representation.

- Problem: Users want to choose a solution at run-time.
- Old Solution: Use **`if-statement`** or **`switch statement`** whenever the choice needs to be made.
- Solution by the Builder pattern: Provide a uniform interface to construct (instantiate) a concrete class.



Builder pattern: a design pattern



Templates: generic programming

- Parameterization of generic containers and algorithms.

```
template <class T, unsigned D1, unsigned D2> TinyMat;
TinyMat<double,3,3> Strain[10];
```

- Standard Template Library (STL) provides a variety containers, iterators and algorithms.
- Concrete objects are instantiated at compile time: reduce run-time overhead of virtual functions via inheritance.
- High optimization can be achieved by using inline functions and inline operators.
  - Special cases (partial specialization) can be specifically written for optimization and linked to existing libraries.
  - Using inline and references, a series of operations can be performed without creating any temporary copies : Expression Templates.

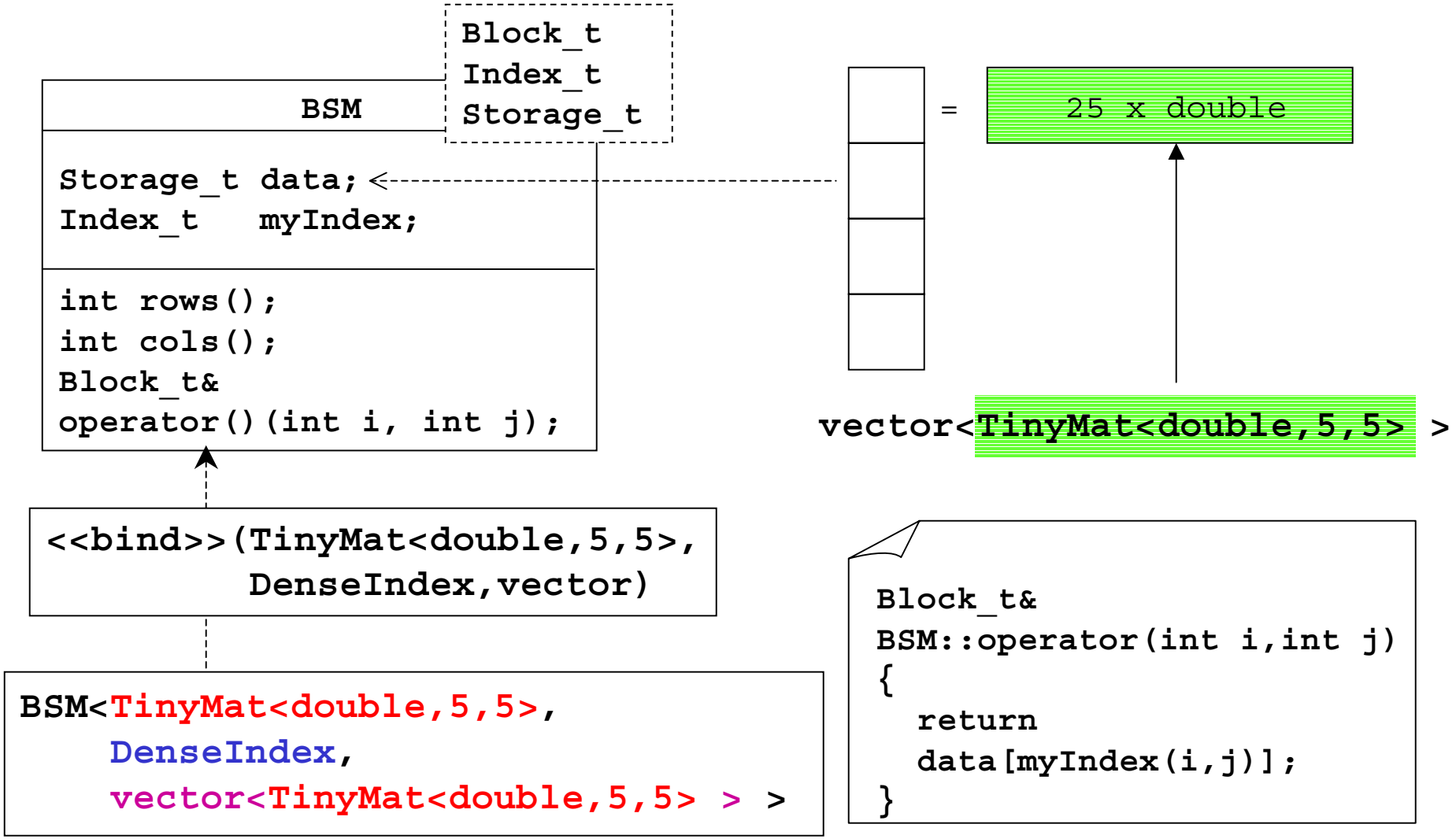
BlockSparseMatrix: example of generic programming

- Requirements:
  - Sparse matrices are frequented used in materials simulations: e.g., localized orbital basis functions of a finite overlap range.
  - Operations using **N x N** block matrices improve performance.
  - Sparse index scheme: a fixed/changing index?
    a index for fast assignment/fast access?
  - How to store a set of block matrices.

- A solution by a templated class:
  - Type of a block matrix (`Block_t`)
  - Index type (`Index_t`)
  - Storage of blocks (`Storage_t`)

```
template<class Block_t,
         class Index_t,
         class Storage_t>
class BSM;
```

BlockSparseMatrix: example of generic programming



Parallel programming and OO frameworks

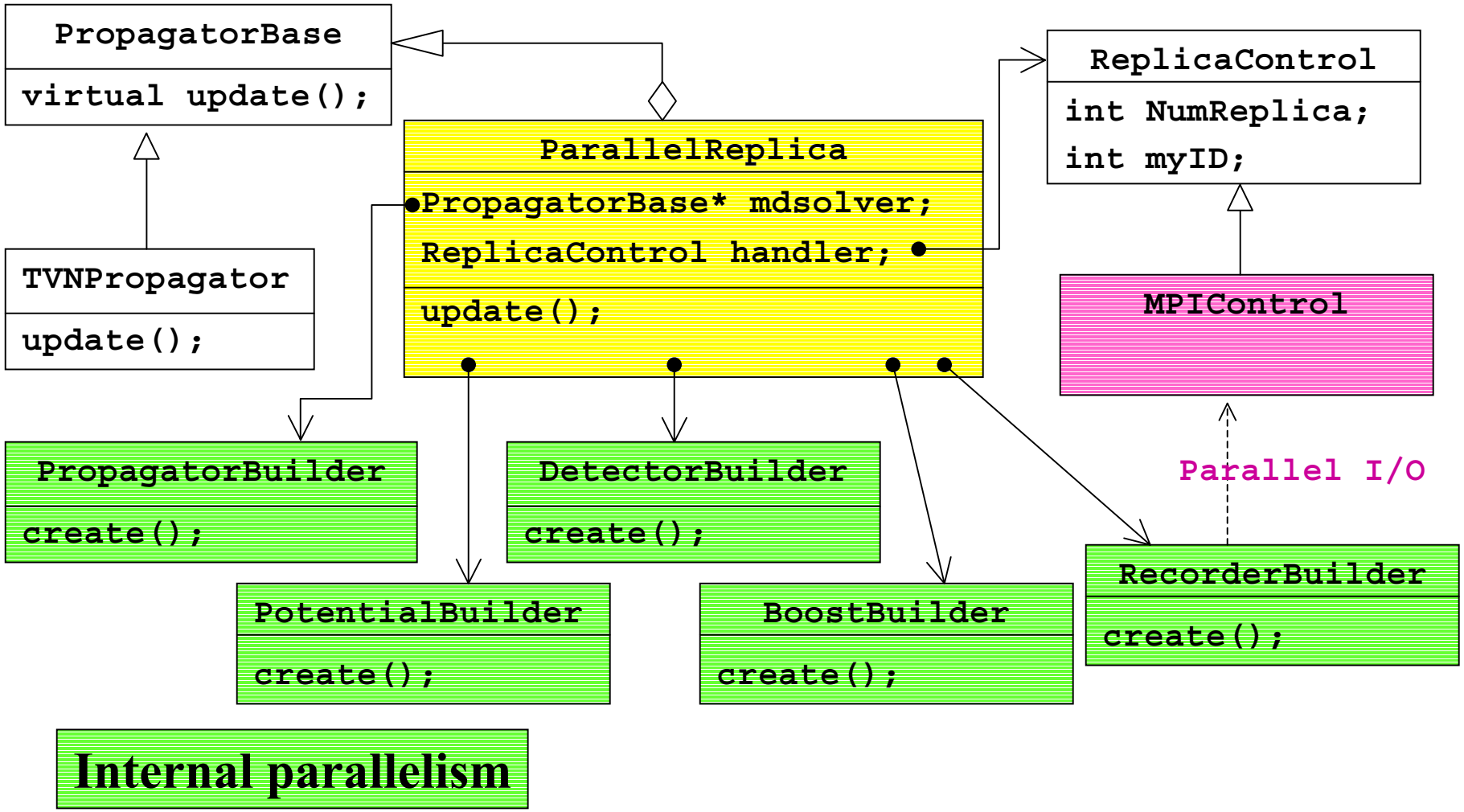
- Parallel programming is essential in many simulations.
- Common parallel libraries: pthreads, OpenMP, MPI …
  - Heterogeneous memory hierarchy of current parallel computers (e.g., linux clusters) can be optimally utilized by combinations.
- Low-level parallelism → generic container, e.g., **`MPIVector`**.
  - Demands high optimization on a single processor.
  - Demands minimum run-time overhead.
  - Requires thread-safe implementations.
- Hi-level parallelism → algorithms
  - Bases on efficient low-level serial/parallel containers.
  - Uses generic parallel engine to hide underlying parallelism.

Parallel OO programming: parallel-replica dynamics

- The parallel-replica method extends the time scale of rare events by running a number of replicas.*
- At compile time, a user chooses parallel libraries
  - for communications between the master and replicas.
  - for containers of objects representing a replica.
- At run time, a user can choose:
  - The number of replicas.
  - Inter-atomic potential type.
  - Detector type to determine a transition: a simple blocking scheme, other automatic procedures.
  - Boost scheme when a coherent structure is established: Voter's boost scheme based on the Hessian, a simple bias potential, or no boost.
  - Recorder for simulations: HDF, ASCII, or other formats.

*A. F. Voter, Phys. Rev. B **57**, R12984 (1998); Lectures by Germann.

Parallel OO programming: parallel-replica dynamics



Why Object-Oriented frameworks?

- Object-oriented programming facilitates development of flexible, easy-to-use and efficient programs
  - Decouple implementations from interfaces → integrity of computational modules can be maintained.
  - Enable large-scale software development while minimizing errors by distributing responsibility among developers .
  - Achieve efficiency under diverse computing environments by using optimized components and libraries.
- Requirements of manageable OO framework development.
  - More time at the design stage is required to develop codes than procedural or modular programming.
  - Discipline to adhere to correct programming habits.
  - Discipline to document designs and implementations.
  - Motivation to learn new algorithms and techniques.

Recommended readings

- Design Patterns: Elements of reusable Object-Oriented Software, Gamma *et al.*, Addison-Wesley, 1995.
- Large-Scale C++ Software Design, Lakos, Addison-Wesley, 1996.
- The C++ Programming Language Special Edition, Stroustrup, Addison-Wesley, 2000.
- Effective C++, 2nd Edition, Meyer, Addison-Wesley, 1997.
- STL Tutorial and Reference Guide, Musser *et al.*, Addison-Wesley, 2nd Edition, 2001.

- **`www.acl.lanl.gov/software`**
- **`oonumeric.org`**
- **`www.physics.ohio-state.edu/~jnkim`**

OO programming: example of a vector class

Vector.h

```
class Vector {
  double *d_data;
  int     d_size;
public:
  Vector();
  Vector(const Vector&);
  ~Vector() {
   if(d_data) delete [] d_data;
  }

  void resize(int num);

  inline int size() const
  {return d_size;}

  double // returns a value
  operator[](int i) const
  { return d_data[i]; }

  double& // assign a value
  operator[](int i)
  { return d_data[i];}

     complete interfaces
};
```

Data of the class

Procedures on the data

```
#include "Vector.h"

int main(int argc, char **argv) {

  Vector A,B; //create vectors

  int n = 10;


  A.resize(n);//calling resize of A
  B.resize(n);//calling resize of B

  for(int i=0; i<A.size(); i++)
    A[i] = random();

  Vector C(A);//creating a vector

  for(int i=0; i<A.size(); i++)
    B[i] = A[i] + C[i];
}
```

What are the objects?

A, B and C of Vector Type

Class diagrams : communicating OO designs