CS 247: Software Engineering Principles

Design Patterns (Composite, Iterator)

Reading: Freeman, Robson, Bates, Sierra, Head First Design Patterns, O'Reilly Media, Inc. 2004

Ch 9: Composite and Iterator Patterns

Electronic text available from UW Library Web site

Today's Agenda

Design patterns: codified solutions that put design principles into practice, to improve the modularity of our code.

OO Ba

- Separation of Concerns
- Encapsulate what is lik
- Encapsulate Data Repr
- Abstraction (interfaces,
- Reuse (through compo
- Polymorphism

OO Principl

- Open Closed Principle
- Favour Composition over Inhe
- Single Responsibility Principle
- Dependency Inversion Princip
- Liskov Substitutability Principle
- Law of Demeter

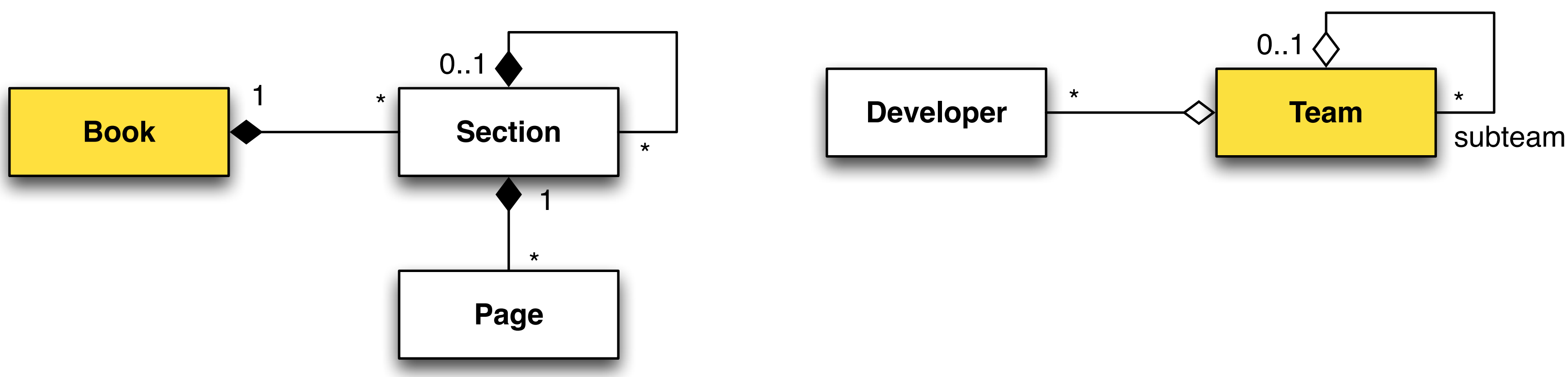
Design Patterns

- Strategy
- Template Method
- Adaptor
- Facade
- Observer
- Model-View-Controller (MVC)
- Composite
- Iterator

Review: Object Composition

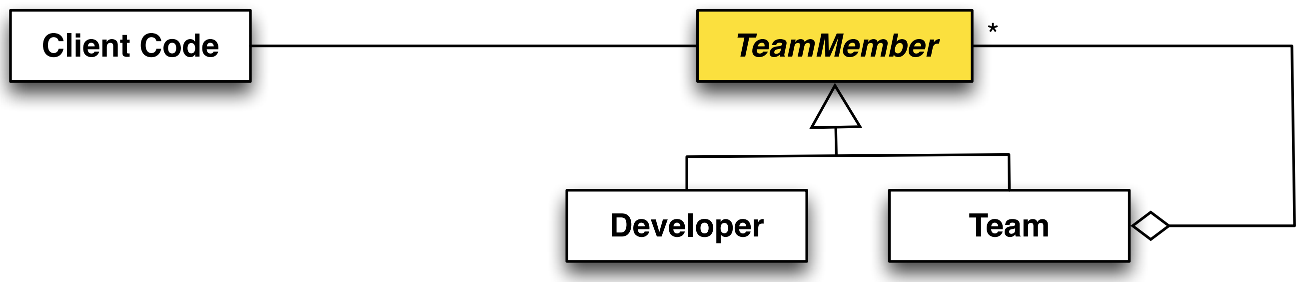
A compound object represents a composition of heterogeneous, possibly recursive, component objects

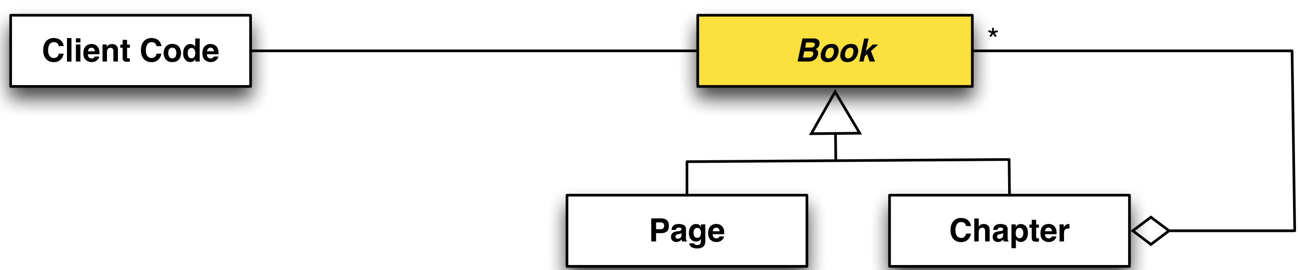
Law of Demeter: client code interacts with compound object



Composite Design Pattern (Idea)

The Composite Pattern takes a different approach: gives the client access to all member types in a compound object via a uniform interface.





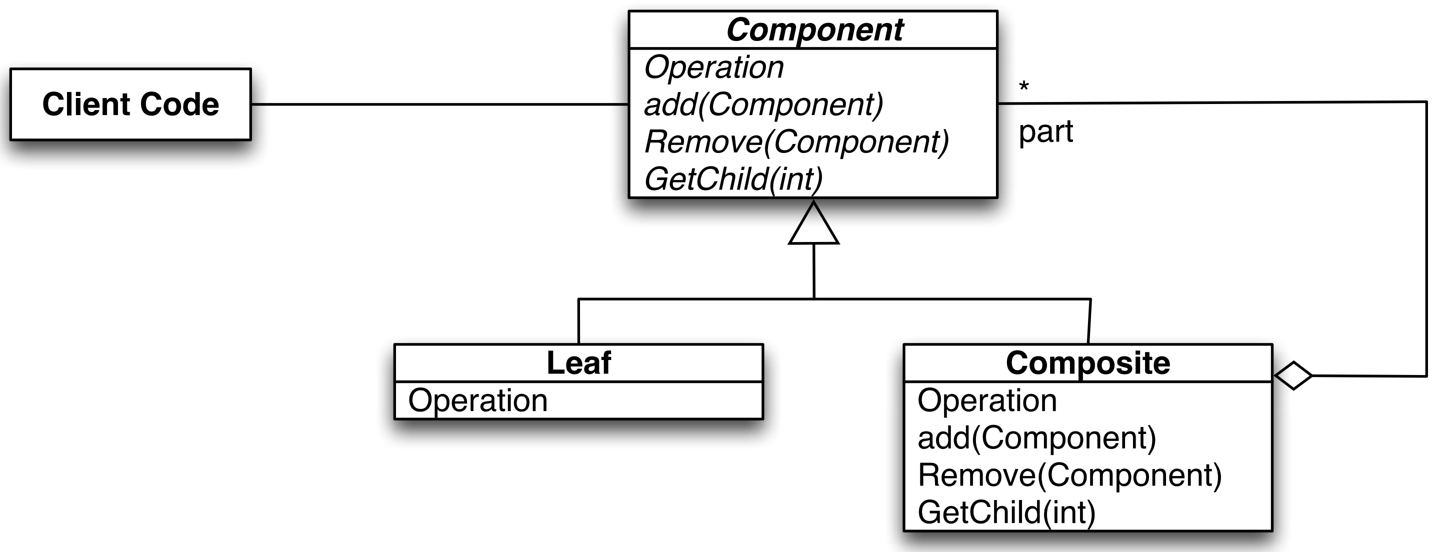
Composite Pattern

Problem: composite object consists of several heterogenous parts

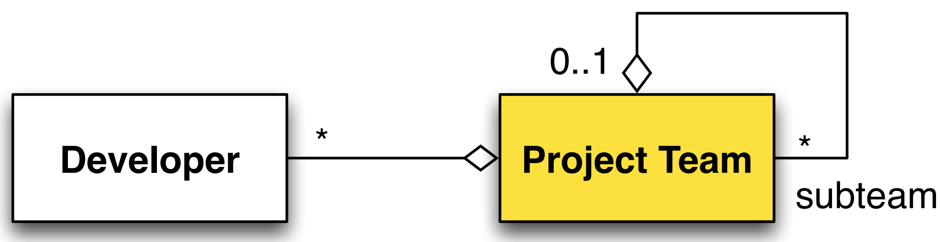
- Client code is complicated by knowledge of object structure
- Client must change if data structure changes

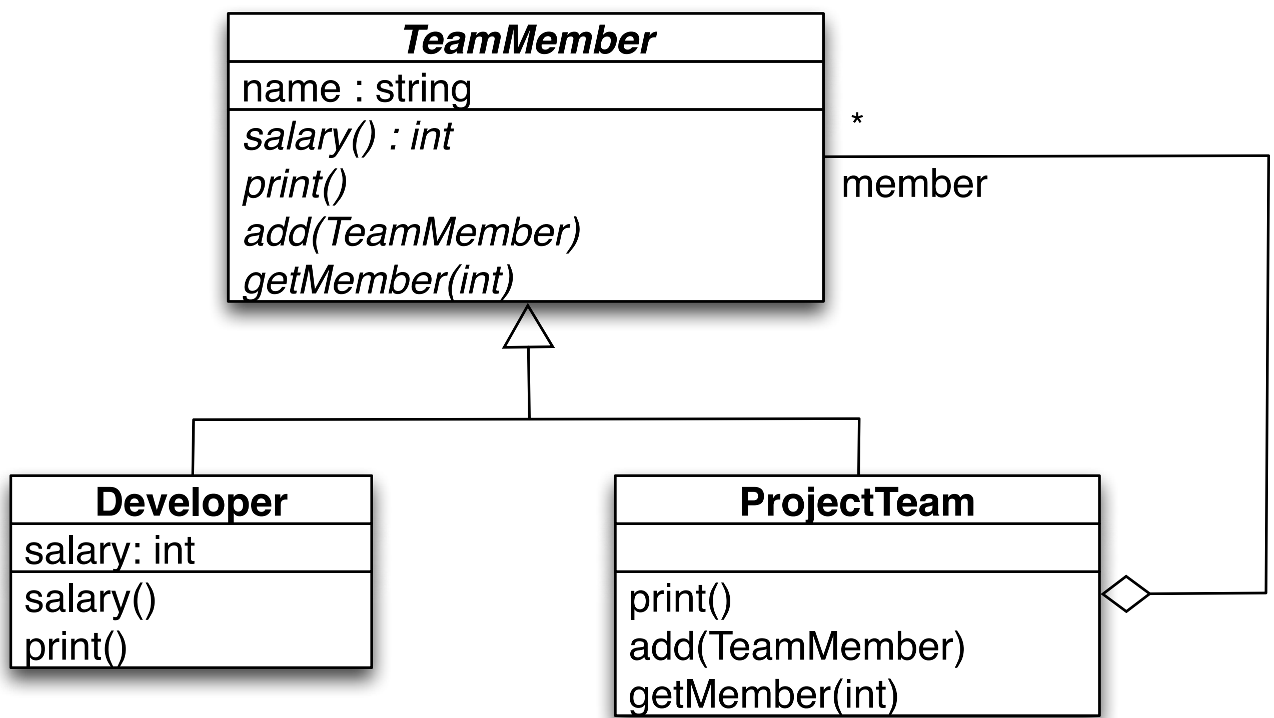
Solution: create a uniform interface for the object's components

- Interface advertises all operations that components offer
- Client deals only with the new uniform interface
- Uniform interface is the union of the components' servcies

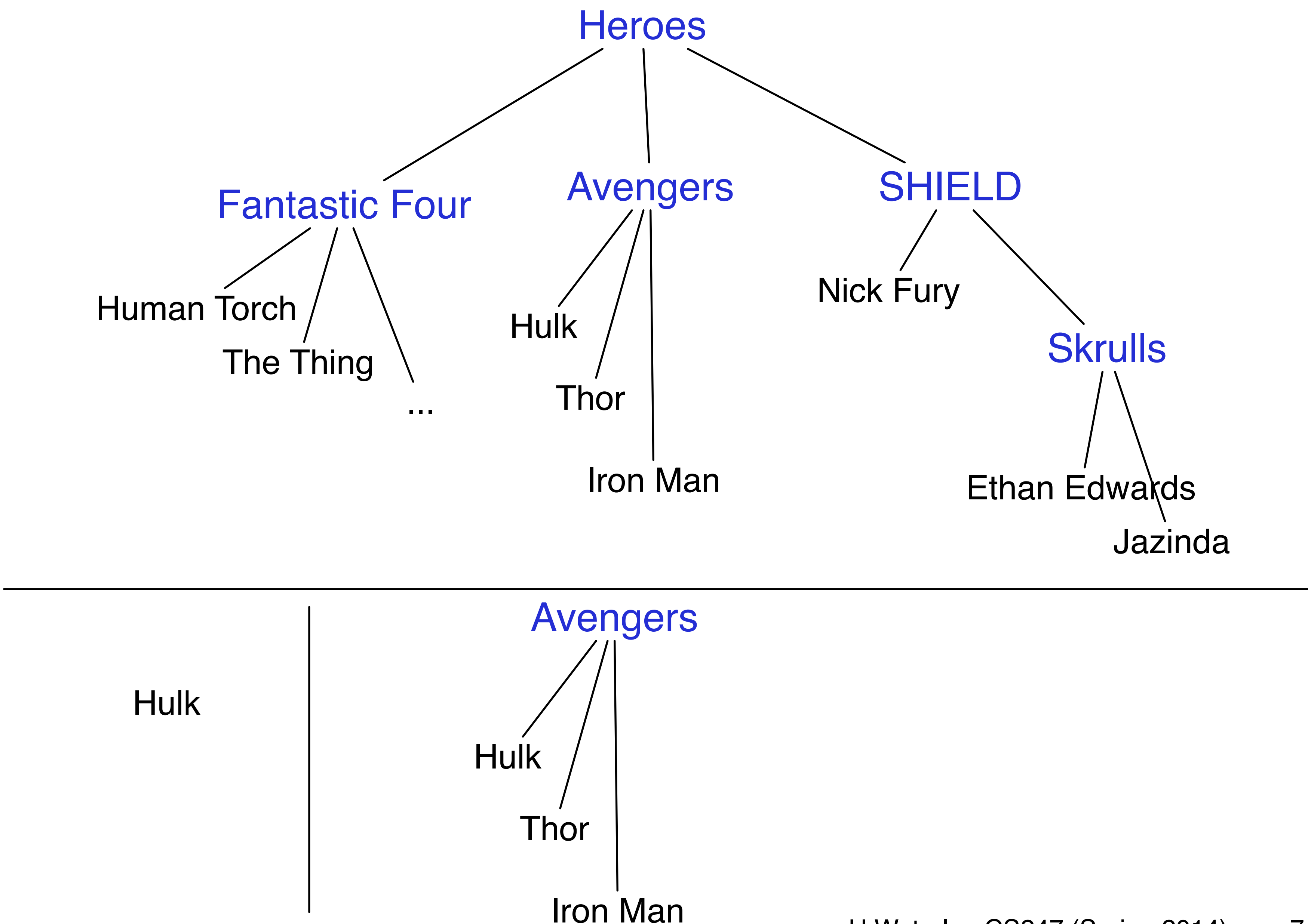


Example


0..1
Developer
*
Project Team
*
subteam


TeamMember
name : string
salary() : int
print()
add(TeamMember)
getMember(int)
*
member
Developer
salary: int
salary()
print()
ProjectTeam
print()
add(TeamMember)
getMember(int)

Team Example



Uniform Interface

```
class TeamMember {
public:
   virtual ~TeamMember() {}

   // leaf-only operations
   virtual int salary() const { return 0;}

   // component-only operations
   virtual void add(TeamMember*) { }
   virtual TeamMember* getMember(int) const { return 0;}

   // shared operations
   virtual void print() const { std::cout << name_; }

protected:
   TeamMember( const std::string& name );

private:
   std::string name_;
};
```

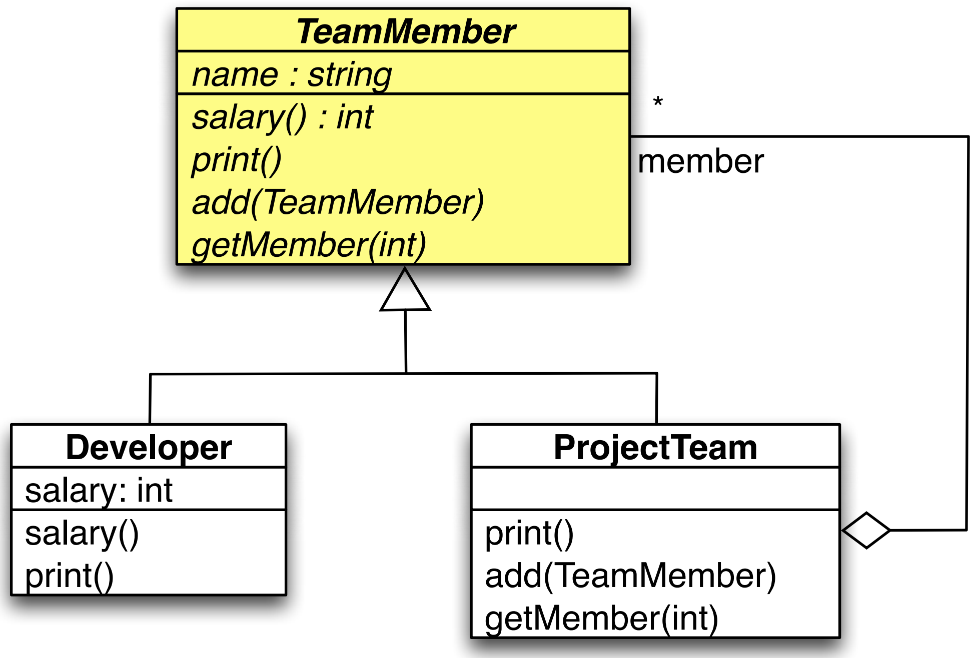
Uniformity vs. Safety

Whether to include component-specific operations in the component interface involves a trade-off between

uniformity - preserving the illusion that component objects can be treated the same way
- promoted by the Composite Pattern

safety - avoiding cases where the client attempts to do something meaningless, like adding components to Leaf objects
- promoted by Liskov Substitutability Principle

Composite Pattern



Consequences:

+ Client deals only with the new uniform interface
+ New leafs and composite types are easy to add
— New operations are harder to add (Visitor Pattern)

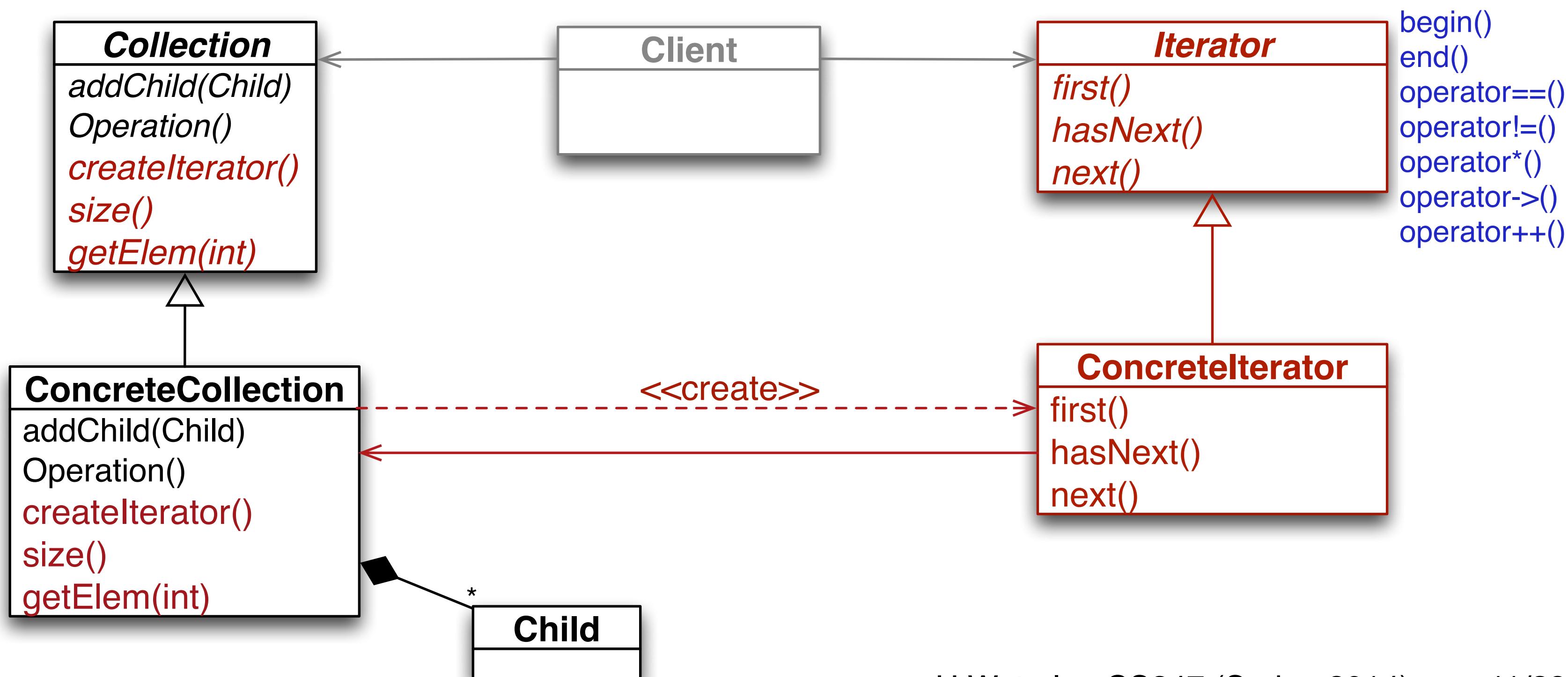
How can client code iterate through a composite object without knowing the composite's structure?

Iterator Pattern

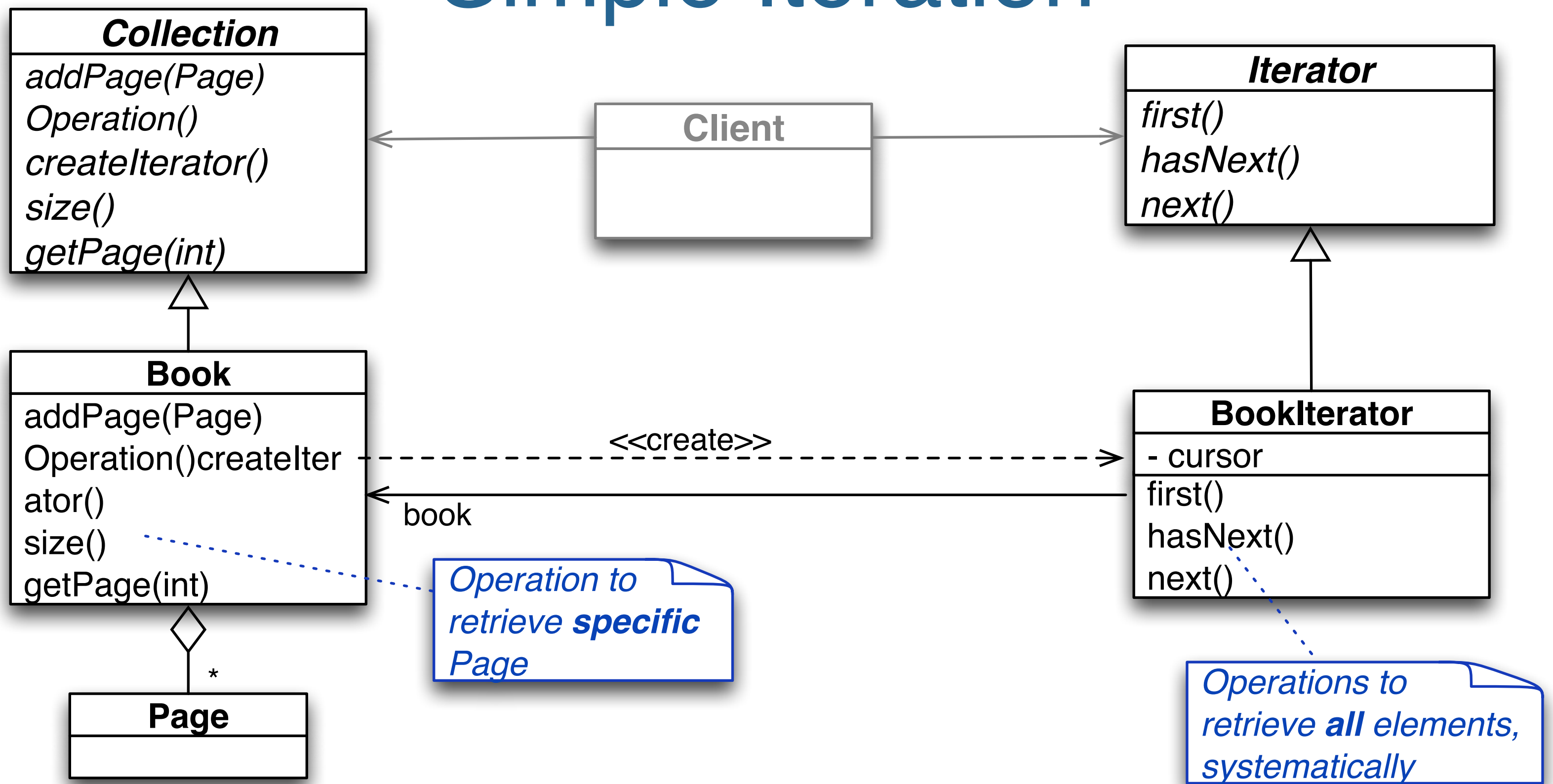
Goals:

(1) To encapsulate the strategy for iterating through a composite (so that it can be changed, at run-time).

(2) Allow the client to iterate through a composite without exposing the composite's representation.



Simple Iteration




```
// client code
Book* b = new Book;
...
BookIterator* iter = b->createIterator();

iter->first();
while ( iter->hasNext() ) {
   Page* p = iter->next();
   // process p
}
```

Book

Composite class is augmented with operations to support the Iterator Pattern.

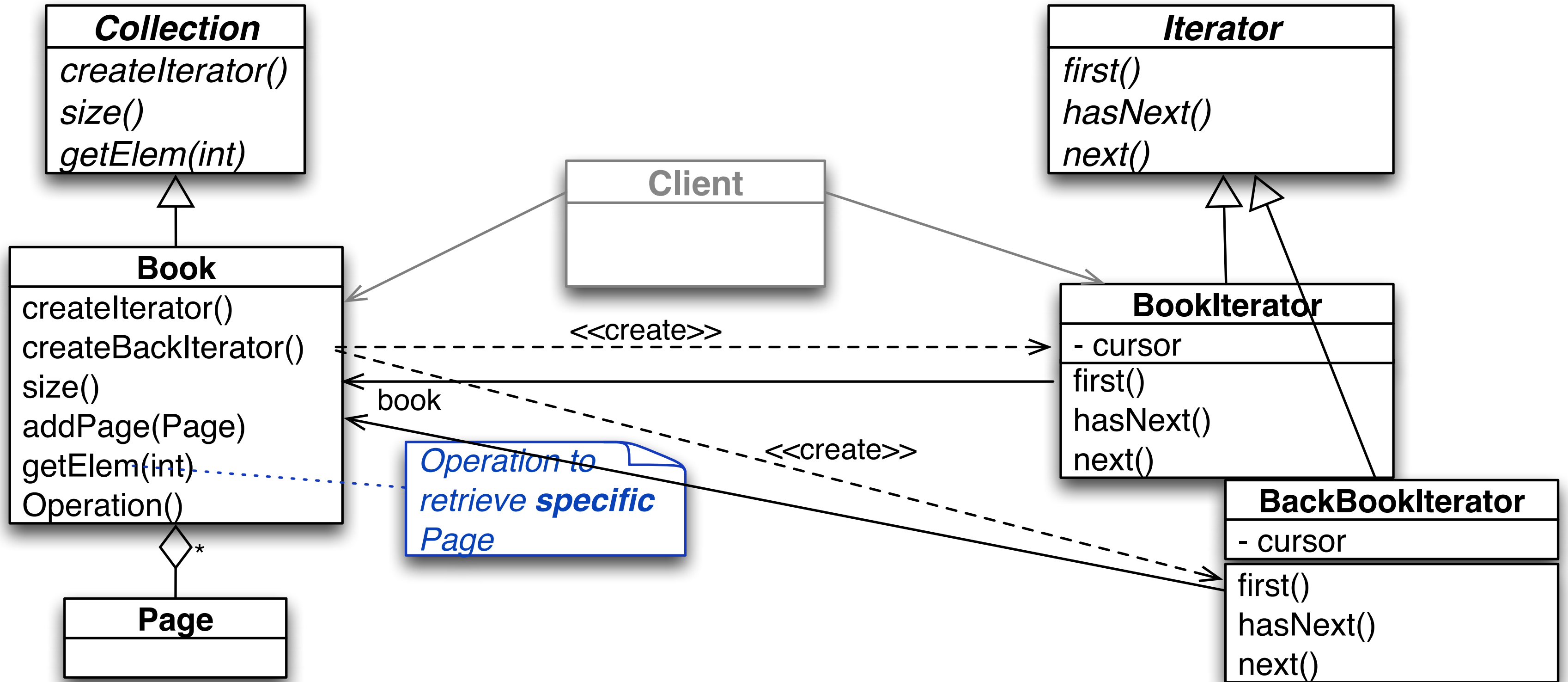
```
class BookIterator;

class Book {
public:
    void addPage(Page*);
    Page* getPage(int) const;
    int size() const;
    BookIterator* createIterator();

private:
    std::vector<Page*> pages_;
};
```

```
BookIterator* Book::createIterator() {
    return new BookIterator(this);
}
```

Book Iterator

```
class BookIterator {
public:
    BookIterator(Book* b) : book_(b), cursor_(0) {}
    Page* next();
    bool hasNext() const;
    void first() { cursor_ = 0; }
private:
    Book* book_;
    int cursor_;
};

bool BookIterator::hasNext() const {
    return cursor_ < book_->size();
}

Page* BookIterator::next() {
    if (!hasNext()) {
        return NULL;
    }
    Page* result = book_->getPage(cursor_);
    cursor_++;
    return result;
}
```

Simple Iteration




```
// client code
Book* b = new Book;
...
BookIterator* iter = b->createBackIterator();

iter->first();
while ( iter->hasNext() ) {
   Page* p = iter->next();
   // process p
}
```

Book (Revisited)

Composite class is augmented with operations to support the Iterator Pattern.

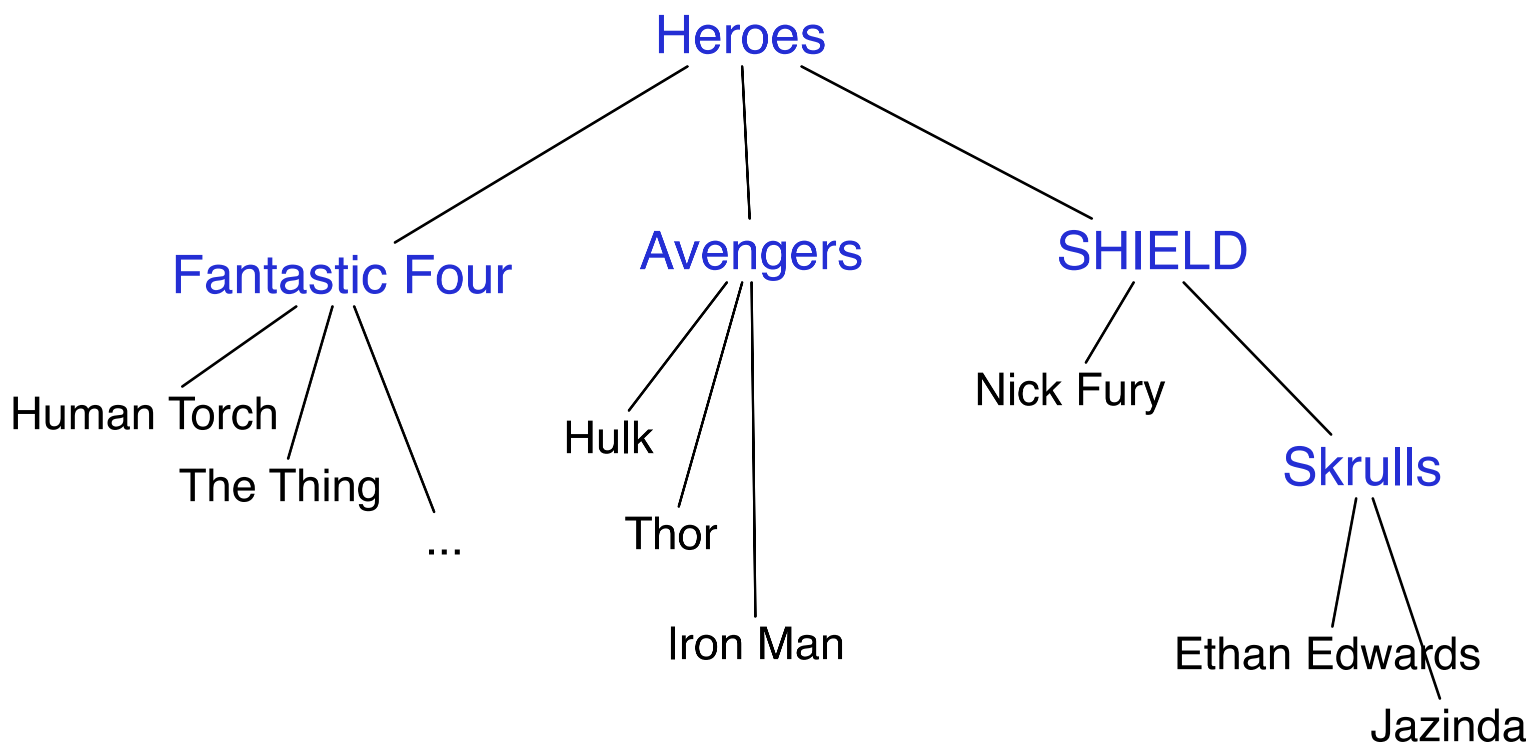
```
class BackBookIterator;

class Book {
public:
    void addPage(Page*);
    Page* getPage(int) const;
    int size() const;
    BookIterator* createIterator();
    BackBookIterator* createBackIterator();

private:
    std::vector<Page*> pages_;
};



BackBookIterator* Book::createBackIterator() {
    return new BackBookIterator(this);
}
```

Backwards Book Iterator

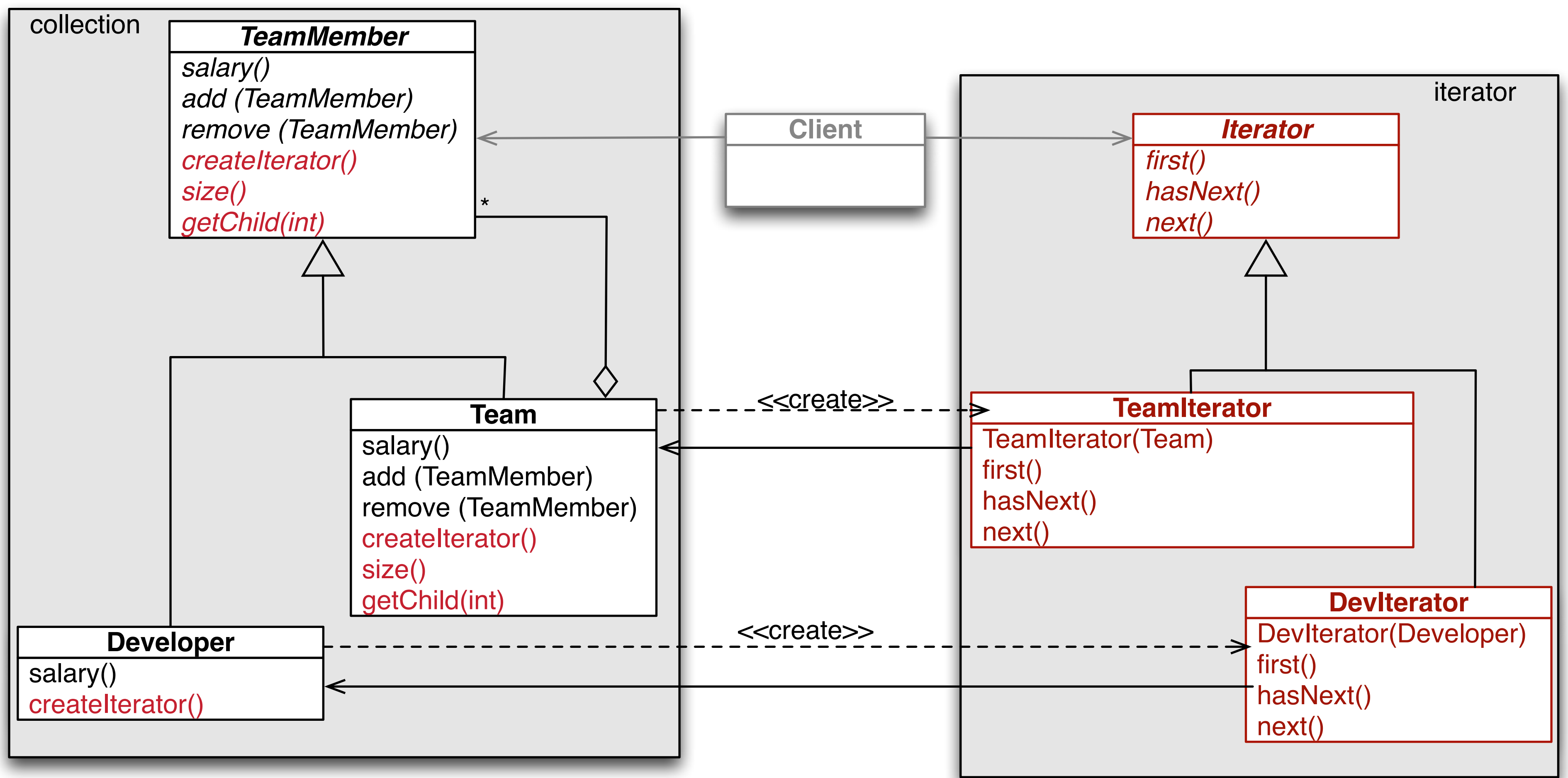
```
class BackBookIterator {
public:
    BackBookIterator(Book* b) : book_(b), cursor_(book_->size()-1) {}
    Page* next();
    bool hasNext() const;
    void first() { cursor_ = book_->size()-1; }
private:
    Book* book_;
    int cursor_;
};

bool BackBookIterator::hasNext() const {
    return cursor_ <= 0;
}

Page* BackBookIterator::next() {
    if (!hasNext()) {
        return NULL;
    }
    Page* result = book_->getPage(cursor_);
    cursor_--;
    return result;
}
```

Iteration over a Composite Object

The more interesting case is when the aggregate is a composite object, in which case we need to construct an Iterator that understands and navigates the composite.



Composite Iteration



Client Code

Iterate through all members in the composite.

```
TeamMember* employees = new Team (...
Iterator* iter = employees->createIterator();
                                            // Team Iterator
iter->first();
while ( iter->hasNext() ) {
  TeamMember* m = iter->next();
  // process member m
}
```

Iterate through all members in a leaf (not very interesting).

```
TeamMember* employees = new Developer (...
Iterator* iter = employees->createIterator();
                                             // Dev Iterator
iter->first();
while ( iter->hasNext() ) {
  TeamMember* m = iter->next();
  // process member m
}
```

Create Iterator

Each concrete subclass in the composite knows how to create its own corresponding Iterator.

```
Iterator* Developer::createIterator() {
  return new DevIterator(this);
}
```

```
Iterator* Team::createIterator() {
  return new TeamIterator(this);
}
```

Developer Iterator

```
class DevIterator : public Iterator {
private:
  Developer* dev_;
  Developer* cursor_;
public:
  DevIterator(Developer* dev) : dev_(dev), cursor_(dev) {}
  virtual void first() { cursor_ = dev_; }
  virtual bool hasNext() { return (cursor_ != 0); }
  virtual TeamMember* next();
};

TeamMember* DevIterator::next() {
  if ( hasNext() ) {
    cursor = 0;
    return dev_;
  }
  return 0;
}
```

Team Behaviour

The Composite objects contribute to iteration with operations to retrieve child elements.

```
class Team : public TeamMember {
private:
  std::vector<TeamMember*> members_;
public:
  ...
  virtual void add( TeamMember* newMember ) {
      members_.push_back( newMember ); }
  virtual int size() const { return members_.size(); }
  virtual TeamMember* getChild(int i) const {
      return members_.at(i); }
};
```

Team Iterator

Each composite node maintains a collection of child nodes.
As the composite iterator walks through the tree, it

- keeps an iterator (cursor) for each node along partially searched path
- puts iterators on stack as the nodes are encountered (DFS)

```
class TeamIterator : public Iterator {
private:
   TeamMember* members_;            // pointer to composite
   struct IterNode;                 // < node, cursor>
   std::stack<IterNode*> istack;    // stack of iterators

public:
  TeamIterator(TeamMember* m) : members_(m) { first(); }

  virtual void first();         // initialize Iterator stack
  virtual bool hasNext();
  virtual TeamMember* next();
};
```

TeamIterator::first()

```
struct TeamIterator::IterNode {
  TeamMember *node_;
  int cursor_;   // ranges from -1 .. node_->size()

  IterNode(TeamMember *m) : node_(m), cursor_(-1) {}
};
```

Initialize the iterator stack with a cursor for the whole composite.

```
void TeamIterator::first() {
  while ( !istack.empty() ) {
    istack.pop();
  }

  istack.push( new IterNode( members_ ) );
}
```

TeamIterator::next()

```
TeamMember* TeamIterator::next() {  // preorder iteration
  if ( hasNext() ) {   // have cursors reached their limit?
    IterNode* top = istack.top();
    istack.pop();

    // cursor points to node (could be Developer or Team)
    if (top->cursor_==-1) {
      top->cursor_ += 1;
      istack.push(top);  // advance cursor to first child
      return top->node_; // return node
    }

    // cursor points to one of the node's children
    TeamMember *elem = top->node_->getChild(top->cursor_);
    top->cursor_ += 1;
    istack.push(top);     // advance cursor to next child
    istack.push(new IterNode(elem)); // push new cursor
    return next();        // recurse
  }
  else return 0;
}
```

TeamIterator::hasNext()

Check if stack contains an iterator that has not retrieved all children of its respective node

```
bool TeamIterator::hasNext() {
  while ( !istack.empty() )  {

    Iter *top = istack.top();
    if ( top->cursor_ < top->node_->size() ) {
      return true;
    }
    istack.pop();
    delete top;
  }

  return false;
}
```

Summary

The goal of design patterns is to encapsulate change

Composite Pattern: encapsulates the structure of a heterogeneous, possibly recursive data structure

Iterator Pattern: encapsulates the iteration of a heterogeneous, possibly recursive data structure