Design Patterns

Claus Jensen

What is a Design Pattern?

- A design pattern
  - Abstracts a recurring design structure
  - Distils design experience
  - Promotes reuse of design and code
  - Gives an opportunity to see how the expert designs
  - Provide a *vocabulary* for talking about design

Background

- Software design patterns are based (somewhat) on work by the architect Christopher Alexander
- Gamma, Helm, Johnson, and Vlissides (the “Gang of Four”) – Design Patterns, Elements of Reusable Object-Oriented Software

Structure of a pattern

- Name
- Intent
- Motivation
- Applicability
- Structure
- Consequences
- Implementation
- Known Uses
- Related Patterns

Some design patterns

- Bridge
- Factory method
- Iterator
- Proxy
- Strategy
- Command

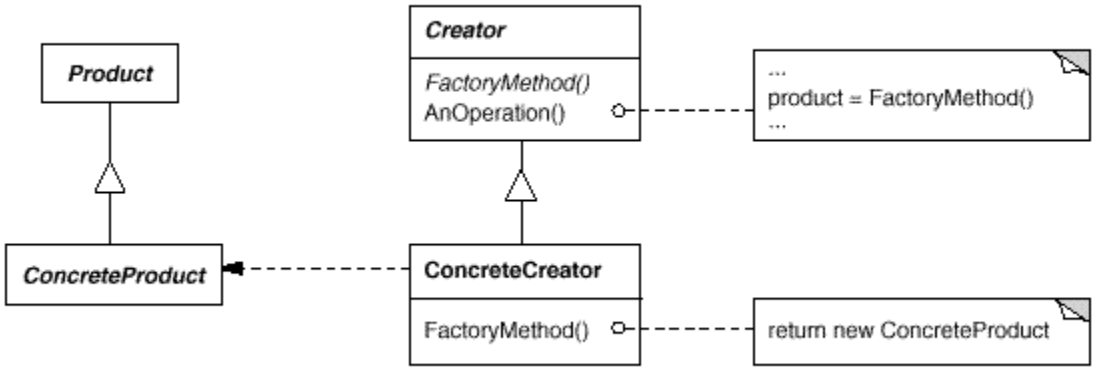
Factory method

- **Intent**
  - Define an interface for creating an object, but let subclasses decide which class to instantiate.

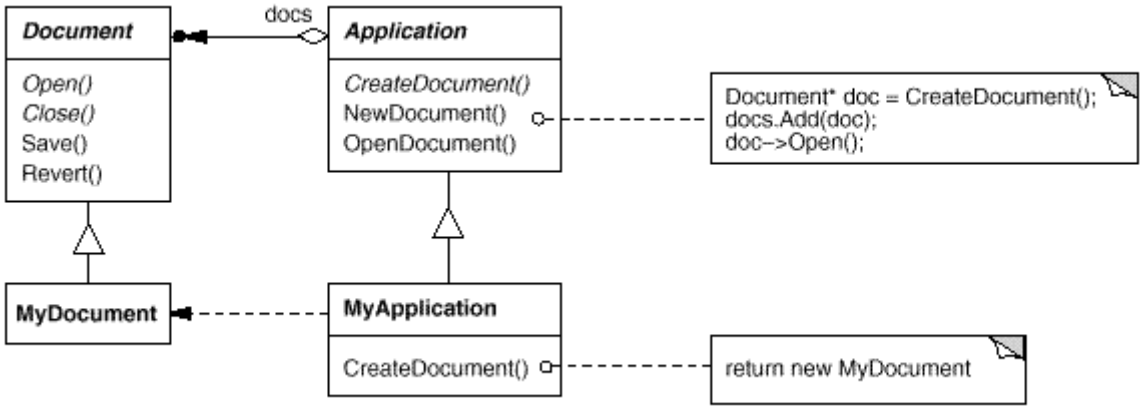
Factory method

- **Motivation**
  - Consider a framework for applications that can present multiple documents to the user. Two key abstractions in this framework are the classes Application and Document. Both classes are abstract, and clients have to subclass them to realize their application-specific implementations.
  - Because the particular Document subclass to instantiate is application-specific, the Application class can't predict the subclass of Document to instantiate—the Application class only knows *when* a new document should be created, not *what kind* of Document to create.

Factory method



Factory method



Factory method

- **Applicability**
  - When a class can't anticipate the class of objects it must create.
  - When a class wants its subclasses to specify the objects it creates.
  - When classes delegate responsibility to one of several helper subclasses, and you want to localize the knowledge of which helper subclass is the delegate.

Factory method

Java code (Hello):

```
public class Person {
    public String name; // name string
    private String gender; // gender : M or F
    public String getName() {
        return name; }
    public String getGender() {
        return gender; }
}
```

Factory method

```
public class Male extends Person {
    public Male(String fullName) {
        System.out.println("Hello Mr. "+fullName);}
}
public class Female extends Person {
    public Female(String fullNname) {
        System.out.println("Hello Ms. "+fullNname);}
}
```

Factory method

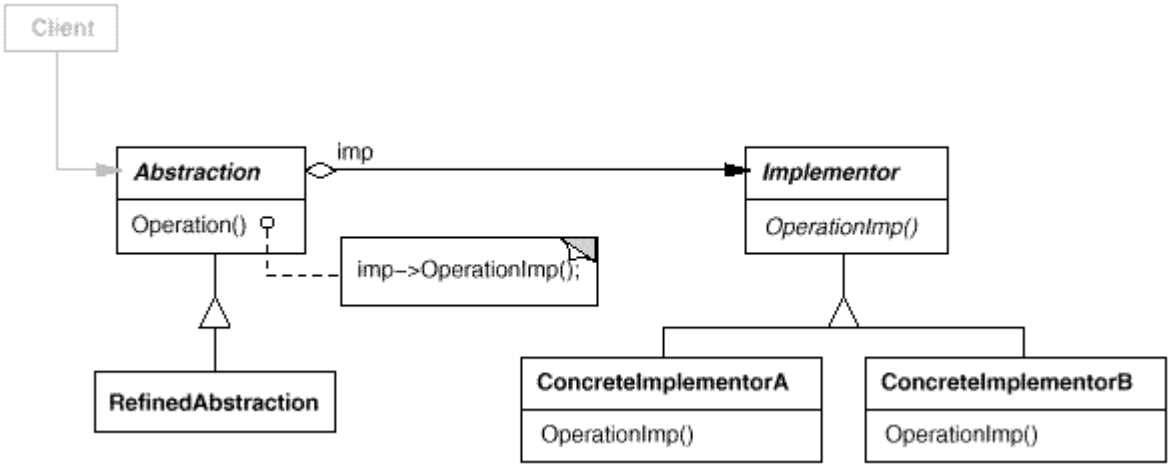
```
public class SalFactory {
    public static void main(String args[]) {
        SalFactory factory = new SalFactory();
        factory.getPerson(args[0], args[1]);
    }
    public Person getPerson(String name, String gender) {
        if (gender.equals("M"))
            return new Male(name);
        else if(gender.equals("F"))
            return new Female(name);
        else
            return null;}
}
```

Java Gamma M -> “Hello Mr. Gamma”

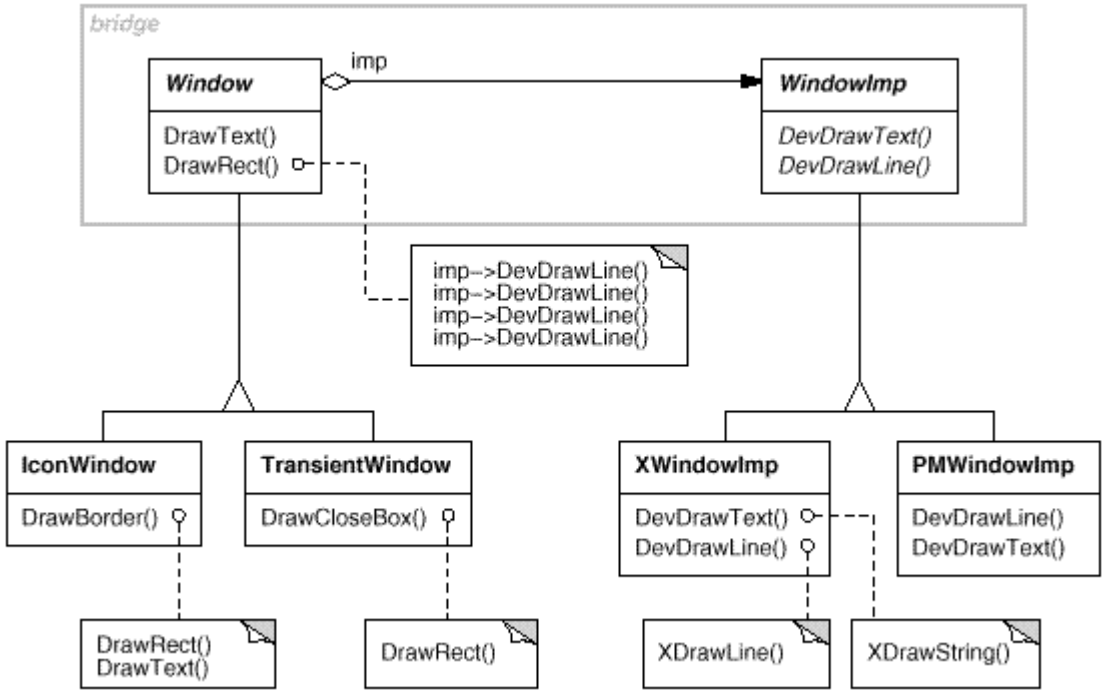
Bridge

- Intent
  - Decouple an abstraction from its implementation so that the two can vary independently.
- Motivation
  - To avoid that inheritance binds an implementation to the abstraction permanently and thereby decrease the flexibility of the design.

Bridge


Client
Abstraction
Operation()
imp
Implementor
OperationImp()
imp->OperationImp();
RefinedAbstraction
ConcreteImplementorA
OperationImp()
ConcreteImplementorB
OperationImp()

Bridge


bridge
Window
DrawText()
DrawRect()
imp
WindowImp
DevDrawText()
DevDrawLine()
imp->DevDrawLine()
imp->DevDrawLine()
imp->DevDrawLine()
imp->DevDrawLine()
IconWindow
DrawBorder()
TransientWindow
DrawCloseBox()
XWindowImp
DevDrawText()
DevDrawLine()
PMWindowImp
DevDrawLine()
DevDrawText()
DrawRect()
DrawText()
DrawRect()
XDrawLine()
XDrawString()

Bridge

- Applicability
  - Want to avoid a permanent binding between an abstraction and its implementation.
  - Both the abstractions and their implementations should be extensible by subclassing.
  - Changes in the implementation of an abstraction should have no impact on clients

Bridge

```
class XWindowImp : public WindowImp {
    public:
        XWindowImp();

        virtual void DeviceRect(Coord, Coord, Coord, Coord);
        // remainder of public interface...
    private:
        // lots of X window system-specific state, including:
        Display* _dpy;
        Drawable _winid;  // window id
        GC _gc;          // window graphic context    };
```

Bridge

- DeviceRect is implemented for X as follows:

```
void XWindowImp::DeviceRect (
        Coord x0, Coord y0, Coord x1, Coord y1
    ) {
            int x = round(min(x0, x1));
            int y = round(min(y0, y1));
            int w = round(abs(x0 - x1));
            int h = round(abs(y0 - y1));
            XDrawRectangle(_dpy, _winid, _gc, x, y, w, h);    }
```

Exercise

- Find concrete examples where the presented patterns can be used.

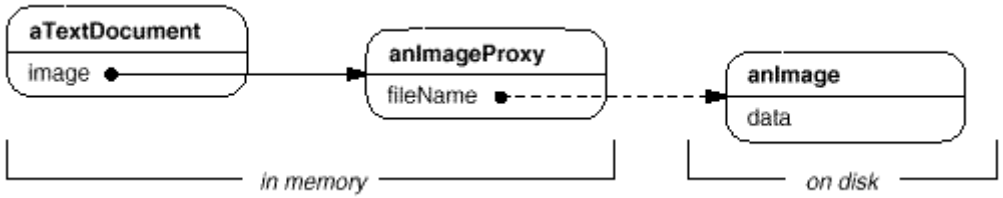
Proxy

- **Intent**
  - Provide a surrogate or placeholder for another object to control access to it.

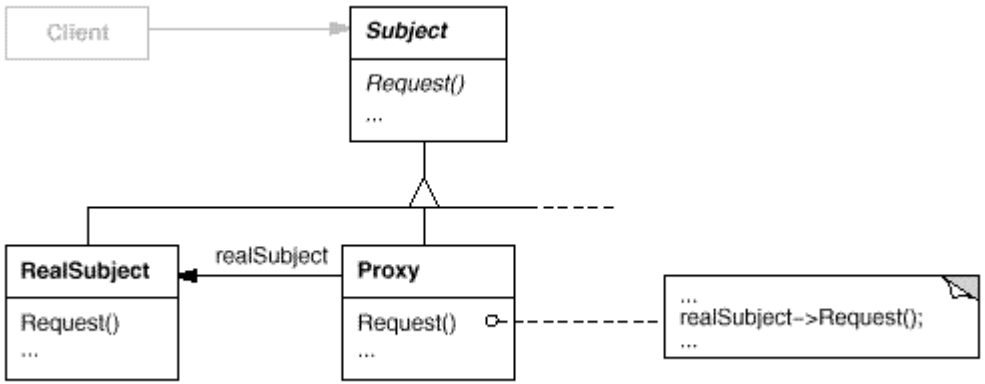
Proxy

- **Motivation**
  - One reason for controlling access to an object is to defer the full cost of its creation and initialization until we actually need to use it. Consider a document editor that can embed graphical objects in a document. Some graphical objects can be expensive to create while opening a document.
  - The solution is to use another object, an image **proxy**, that acts as a stand-in for the real image. The proxy acts just like the image and takes care of instantiating it when it's required.

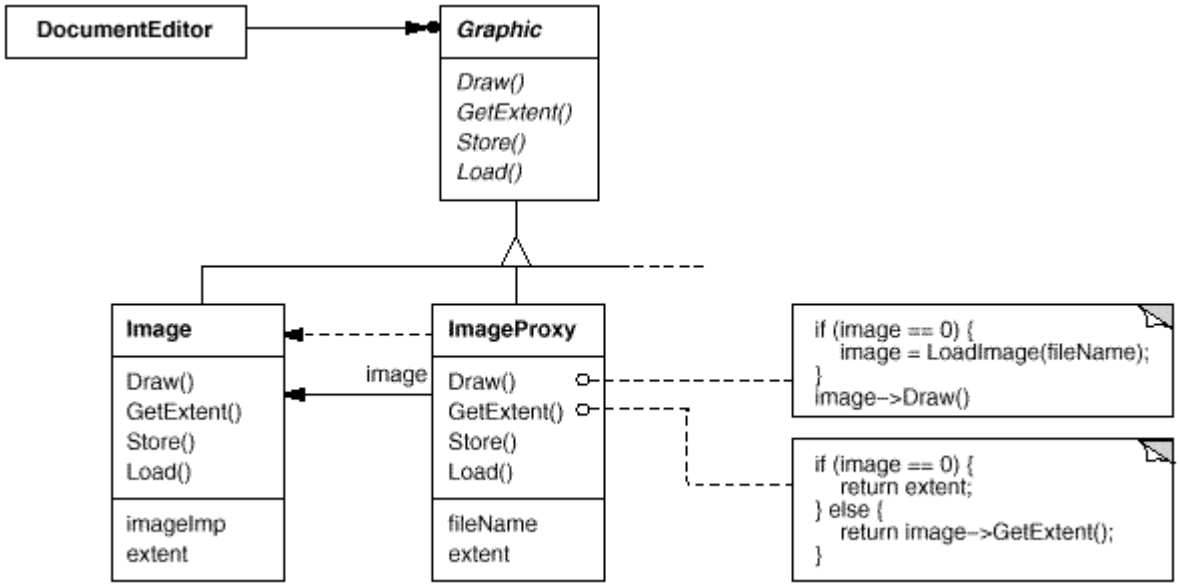
Proxy


aTextDocument
image
anImageProxy
fileName
anImage
data
in memory
on disk

Proxy

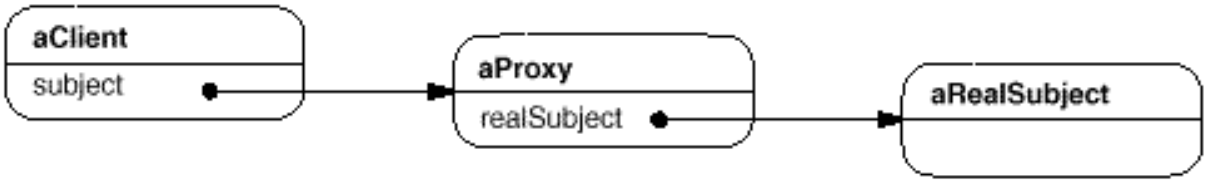

Client
Subject
Request()
...
RealSubject
Request()
...
realSubject
Proxy
Request()
...
...
realSubject->Request();
...

Proxy


DocumentEditor
Graphic
Draw()
GetExtent()
Store()
Load()
Image
Draw()
GetExtent()
Store()
Load()
imageImp
extent
image
ImageProxy
Draw()
GetExtent()
Store()
Load()
fileName
extent
if (image == 0) {
image = LoadImage(fileName);
}
image->Draw()
if (image == 0) {
return extent;
} else {
return image->GetExtent();
}

Proxy

- Here's a possible object diagram of a proxy structure at run-time:



Proxy

```
class ImageProxy : public Graphic {
public:
    ImageProxy(const char* imageFile);
    virtual ~ImageProxy();
    virtual void Draw(const Point& at);
    virtual void HandleMouse(Event& event);
    virtual const Point& GetExtent();
    virtual void Load(istream& from);
    virtual void Save(ostream& to);
protected:
    Image* GetImage();
private:
    Image* _image;
    Point _extent;
    char* _fileName;    };
```

Proxy

```
ImageProxy::ImageProxy (const char* fileName)  {
     _fileName = strdup(fileName);
    _extent = Point::Zero;  // don't know extent yet
    _image = 0;    }

Image* ImageProxy::GetImage() {
    if (_image == 0) {
     _image = new Image(_fileName);
    }
    return _image;    }
```

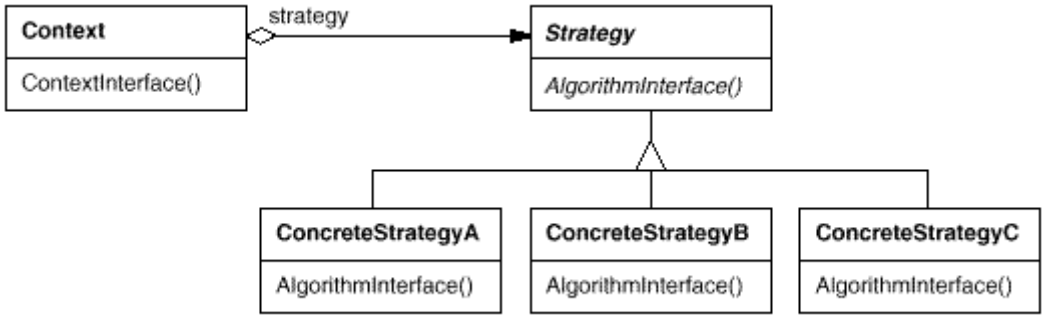
Strategy

- **Intent**
  - Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.

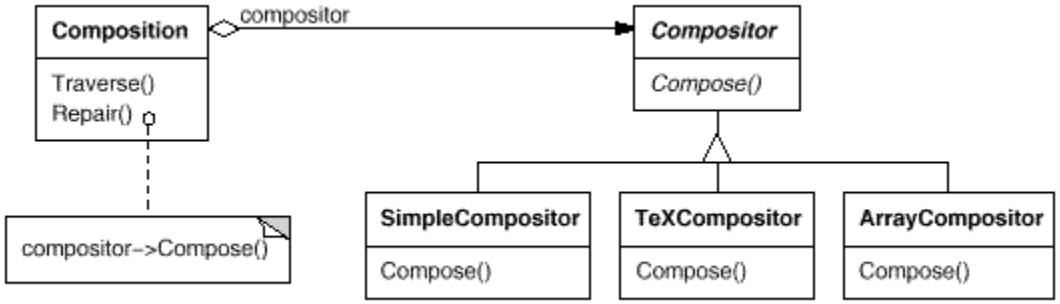
Strategy

- **Motivation**
  - Support multiple algorithms.
  - Different algorithms will be appropriate at different times. We don't want to support multiple algorithms if we don't use them all.
  - It's difficult to add new algorithms and vary existing ones when the algorithm is an integral part of the client.

Strategy


Context
ContextInterface()
strategy
Strategy
AlgorithmInterface()
ConcreteStrategyA
AlgorithmInterface()
ConcreteStrategyB
AlgorithmInterface()
ConcreteStrategyC
AlgorithmInterface()

Strategy


Composition
Traverse()
Repair()
compositor
Compositor
Compose()
compositor->Compose()
SimpleCompositor
Compose()
TeXCompositor
Compose()
ArrayCompositor
Compose()

Strategy

- Suppose a Composition class is responsible the linebreaks of text displayed in a text viewer. Linebreaking strategies aren't implemented by the class Composition, but instead by subclasses of the abstract Compositor class.
- Compositor subclasses implement different strategies:
  - **SimpleCompositor** implements a simple strategy that determines linebreaks one at a time.
  - **TeXCompositor** implements the TeX algorithm for finding linebreaks. This strategy tries to optimize linebreaks globally, that is, one paragraph at a time.
  - **ArrayCompositor** implements a strategy that selects breaks so that each row has a fixed number of items. It's useful for breaking a collection of icons into rows, for example.

Strategy

- **Applicability**
  - When many related classes differ only in their behaviour.
  - When you need different variants of an algorithm. For example, you might define algorithms reflecting different space/time trade-offs.
  - When an algorithm uses data that clients shouldn't know about.

Strategy

```
class Composition {
public:
    Composition(Compositor*);
    void Repair();
private:
    Compositor* _compositor;
    Component* _components;    // the list of components
    int _componentCount;       // the number of components
    int _lineWidth;            // the Composition's line width
    int* _lineBreaks;          // the position of linebreaks in components
    int _lineCount;            // the number of lines    };
```

Strategy

```
 class Compositor {
public:
    virtual int Compose(
            Coord natural[], Coord stretch[], Coord shrink[],
            int componentCount, int lineWidth, int breaks[]
    ) = 0;
protected:
    Compositor();    };
```

Strategy

```
class TeXCompositor : public Compositor {
public:
    TeXCompositor();
    virtual int Compose(
            Coord natural[], Coord stretch[], Coord shrink[],
            int componentCount, int lineWidth, int breaks[]
            );
            // ...
};
```

Strategy

- To instantiate Composition, you pass it the compositor you want to use:

```
Composition* quick = new Composition(new
  SimpleCompositor);
Composition* slick = new Composition(new
  TeXCompositor);
Composition* iconic = new Composition(new
  ArrayCompositor(100));
```

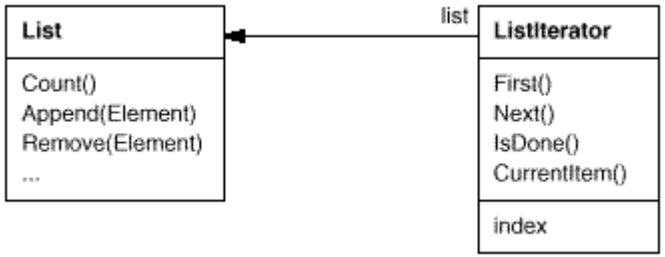
Iterator

- **Intent**
  - Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

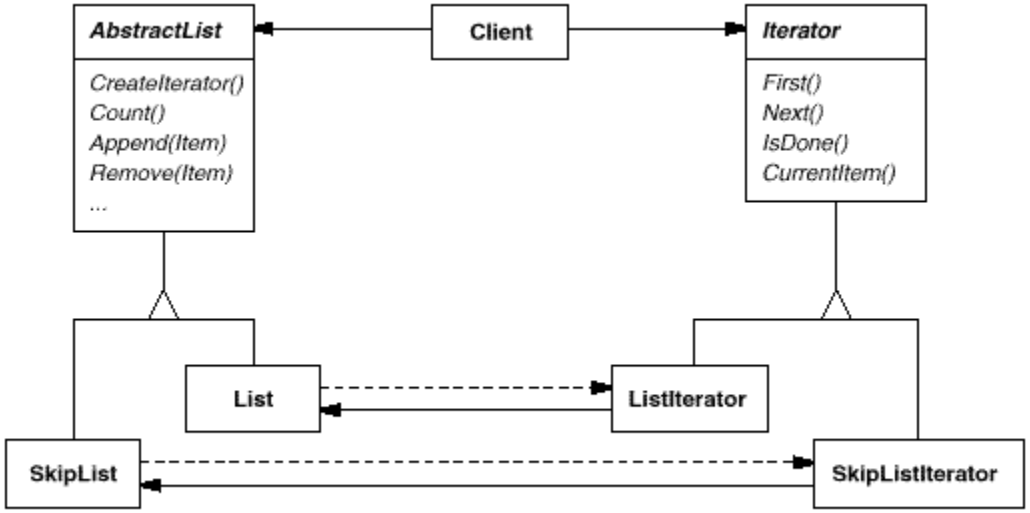
Iterator

- Motivation
  - It should possible to access an aggregate object without exposing its internal structure (for example a lists elements).
  - It should also be possible to traverse the list in different ways.

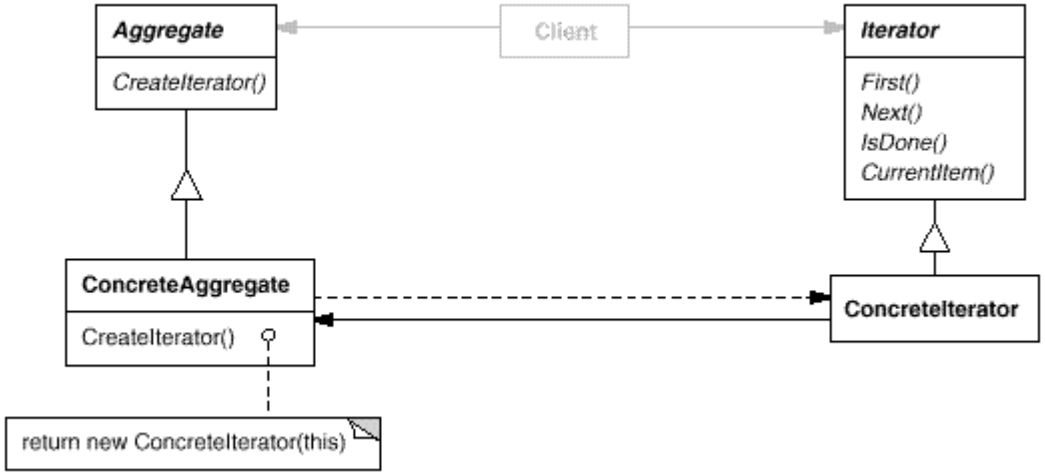
Iterator


List
Count()
Append(Element)
Remove(Element)
...
list
ListIterator
First()
Next()
IsDone()
CurrentItem()
index

Iterator


AbstractList
CreateIterator()
Count()
Append(Item)
Remove(Item)
...
Client
Iterator
First()
Next()
IsDone()
CurrentItem()
List
ListIterator
SkipList
SkipListIterator

Iterator


Aggregate
CreateIterator()
Client
Iterator
First()
Next()
IsDone()
CurrentItem()
ConcreteAggregate
CreateIterator()
ConcreteIterator
return new ConcreteIterator(this)

Iterator

- **Applicability**
  - Want to access an aggregate object's contents without exposing its internal representation.
  - Want to support multiple traversals of aggregate objects.
  - Want to provide a uniform interface for traversing different aggregate structures.

Iterator

```
 template <class Item>
class ListIterator : public Iterator<Item> {
public:
    ListIterator(const List<Item>* aList);
    virtual void First();
    virtual void Next();
    virtual bool IsDone() const;
    virtual Item CurrentItem() const;
private:
    const List<Item>* _list;
    long _current;    };
```

Iterator

```
template <class Item>
ListIterator<Item>::ListIterator (
    const List<Item>* aList
) : _list(aList), _current(0) {    }

template <class Item>
void ListIterator<Item>::Next () {
    current++;    }
```

Iterator

```
void PrintEmployees
  (Iterator<Employee*>& i) {
      for (i.First(); !i.IsDone(); i.Next()) {
          i.CurrentItem()->Print();
      }
}
```