Design Patterns
The Way of the Code Ninja

Devert Alexandre
School of Software Engineering of USTC

Table of Contents

1. Introduction
2. The Iterator
3. The Visitor
4. The Factory
   - The fat constructor problem
   - The many constructors problem
   - Construction as an object
5. Conclusion

Agile

Agile software development requires agile code ...

- Code easy to extend
- Adding by extending, not by rewriting

Design patterns

Design patterns are generic *good* solutions to common problems

- Made through years of experience on many projects
- Mostly language independent
- Introduced in *Design Patterns: Elements of Reusable Object-Oriented Software*

Anti-patterns

Anti-patterns are common *bad* solutions

- We all made those
- They can slow you down
- They can put the project in danger

Object programming

Object programming is a *tool*, not a *solution*

- There are good and bad object-oriented designs
- Patterns and anti-patterns are examples of each

Table of Contents

1. Introduction
2. The Iterator
3. The Visitor
4. The Factory
   - The fat constructor problem
   - The many constructors problem
   - Construction as an object
5. Conclusion

A common design pattern

The *Iterator* is very common in APIs

```
list<Cheese> cheeseShop;
...
list<Cheese>::iterator it = cheeseShop.begin();
for( ; it != cheeseShop.end(); ++it)
  if (!(*it).available())
    cout (*it).name() << " not available" << endl;
```

Listing 1: C++ standard library iterator

A common design pattern

The *Iterator* is very common in APIs

```
LinkedList<Cheese> cheeseShop = new LinkedList<Cheese>();
...
Iterator<Cheese> it = cheeseShop.iterator();
while(it.hasNext()) {
  Cheese cheese = it.next();
  if (!cheese.available())
    System.out.println(cheese.name() + " not available");
}
```

Listing 2: Java standard library iterator

A common design pattern

The *Iterator* is very common in APIs

```
cheeseShop = []
...
for cheese in cheeseShop:
  if not cheese.available():
    print cheese.name(), "not available"
```

Listing 3: Python's iterator

Implementation

A typical interface for an Iterator

```
interface Iterator<T> {
  boolean hasNext();

  T next();
}
```

- *hasNext* ⇒ tell if more elements
- *next* ⇒ give the next element

Goal

An iterator allow to separate the *container* from the *traversal* of that container

1. can abstract the container type (array, tree, hash table)
2. can abstract the traversal algorithm

Abstracting the container

Displaying the non-available cheese

Cheese Shop	
Red Leicester	No
Tilsit	No
Caerphilly	No
Bel Paese	No
Red Windsor	No
Stilton	No
Emmental	No
Liptauer	No
Lancashire	No
Danish Blue	No

```
def showCheeseShop(cheeseShop):
  for cheese in cheeseShop:
    if not cheese.available():
      print cheese.name(), "not available"

cheeseShop = [
  Cheese('tilsit'),
  Cheese('stilton')
]

...

showCheeseShop(cheeseShop)
```

Listing 4: list

Abstracting the container

Displaying the non-available cheese

Cheese Shop	
Red Leicester	No
Tilsit	No
Caerphilly	No
Bel Paese	No
Red Windsor	No
Stilton	No
Emmental	No
Liptauer	No
Lancashire	No
Danish Blue	No

```
def showCheeseShop(cheeseShop):
  for cheese in cheeseShop:
    if not cheese.available():
      print cheese.name(), "not available"

cheeseShop = set()
cheeShop.insert('tilsit')
cheeShop.insert('stilton')

...

showCheeseShop(cheeseShop)
```

Listing 5: tree set

Abstracting the container

Displaying the non-available cheese

Cheese Shop	
Red Leicester	No
Tilsit	No
Caerphilly	No
Bel Paese	No
Red Windsor	No
Stilton	No
Emmental	No
Liptauer	No
Lancashire	No
Danish Blue	No

```
def showCheeseShop(cheeseShop):
  for cheese in cheeseShop:
    if not cheese.available():
      print cheese.name(), "not available"

cheeseShop = {
  'tilsit'  : Cheese(),
  'stilton' : Cheese()
}

...

showCheeseShop(cheeseShop.items())
```

Listing 6: hash table

Abstracting the container

Using iterators allows to change the container type

- Very reusable code
- Open to container type changes

Abstracting the traversal

Many algorithms rely on traversing some sort of trees

- *breadth-first* traversal
- *depth-first* traversal
- *infix* traversal
- *postfix* traversal
- ...

Abstracting the traversal

Searching items in a tree : breadth-first traversal

ϵ

0 1

00 01 10 11

000 001 010 011 100 101 110 111

Abstracting the traversal

Searching items in a tree : breadth-first traversal

Abstracting the traversal

Searching items in a tree : breadth-first traversal

ϵ

0 1

00 01 10 11

000 001 010 011 100 101 110 111

Abstracting the traversal

Searching items in a tree : breadth-first traversal

ϵ

0 1

00 01 10 11

000 001 010 011 100 101 110 111

Abstracting the traversal

Searching items in a tree : breadth-first traversal

ϵ

0 1

00 01 10 11

000 001 010 011 100 101 110 111

Abstracting the traversal

Searching items in a tree : breadth-first traversal

Abstracting the traversal

Searching items in a tree : breadth-first traversal

ϵ

0 1

00 01 10 11

000 001 010 011 100 101 110 111

Abstracting the traversal

Searching items in a tree : breadth-first traversal

Abstracting the traversal

Searching items in a tree : breadth-first traversal

Abstracting the traversal

Searching items in a tree : breadth-first traversal

ϵ

0 1

00 01 10 11

000 001 010 011 100 101 110 111

Abstracting the traversal

Searching items in a tree : breadth-first traversal

ϵ

0 1

00 01 10 11

000 001 010 011 100 101 110 111

Abstracting the traversal

Searching items in a tree : breadth-first traversal

ϵ

0 1

00 01 10 11

000 001 010 011 100 101 110 111

Abstracting the traversal

Searching items in a tree : breadth-first traversal

Abstracting the traversal

Searching items in a tree : breadth-first traversal

ϵ

0 1

00 01 10 11

000 001 010 011 100 101 110 111

Abstracting the traversal

Searching items in a tree : breadth-first traversal

ϵ

0 1

00 01 10 11

000 001 010 011 100 101 110 111

Abstracting the traversal

Searching items in a tree : breadth-first traversal

Abstracting the traversal

Searching items in a tree : depth-first traversal

ϵ

0 1

00 01 10 11

000 001 010 011 100 101 110 111

Abstracting the traversal

Searching items in a tree : depth-first traversal

ϵ

0 1

00 01 10 11

000 001 010 011 100 101 110 111

Abstracting the traversal

Searching items in a tree : depth-first traversal

ϵ

0 1

00 01 10 11

000 001 010 011 100 101 110 111

Abstracting the traversal

Searching items in a tree : depth-first traversal

Abstracting the traversal

Searching items in a tree : depth-first traversal

Abstracting the traversal

Searching items in a tree : depth-first traversal

Abstracting the traversal

Searching items in a tree : depth-first traversal

Abstracting the traversal

Searching items in a tree : depth-first traversal

ϵ

0 1

00 01 10 11

000 001 010 011 100 101 110 111

Abstracting the traversal

Searching items in a tree : depth-first traversal

Abstracting the traversal

Searching items in a tree : depth-first traversal

Abstracting the traversal

Searching items in a tree : depth-first traversal

ϵ

0 1

00 01 10 11

000 001 010 011 100 101 110 111

Abstracting the traversal

Searching items in a tree : depth-first traversal

ϵ

0 1

00 01 10 11

000 001 010 011 100 101 110 111

Abstracting the traversal

Searching items in a tree : depth-first traversal

ϵ

0 1

00 01 10 11

000 001 010 011 100 101 110 111

Abstracting the traversal

Searching items in a tree : depth-first traversal

Abstracting the traversal

Searching items in a tree : depth-first traversal

ϵ

0 1

00 01 10 11

000 001 010 011 100 101 110 111

Abstracting the traversal

Searching items in a tree : depth-first traversal

ϵ

0 1

00 01 10 11

000 001 010 011 100 101 110 111

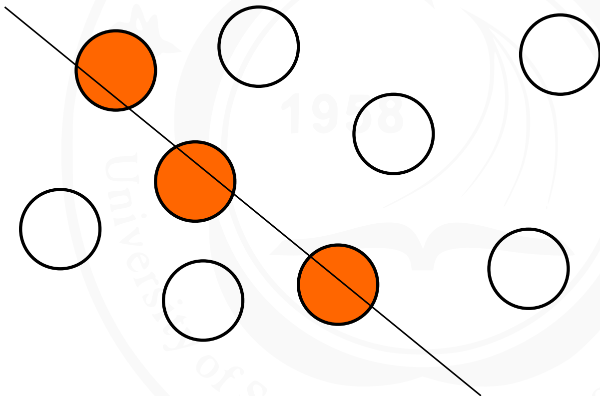
Abstracting the traversal

Each kind of tree traversal can be an iterator

- Easy to change traversal logic
- Isolate the traversal logic

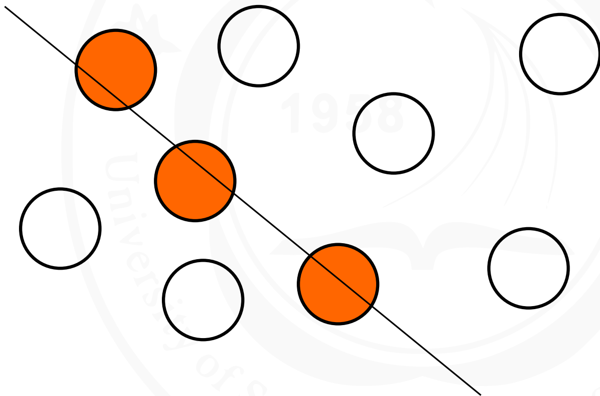
Abstracting the traversal & the container

Finding intersection with a ray and many objects



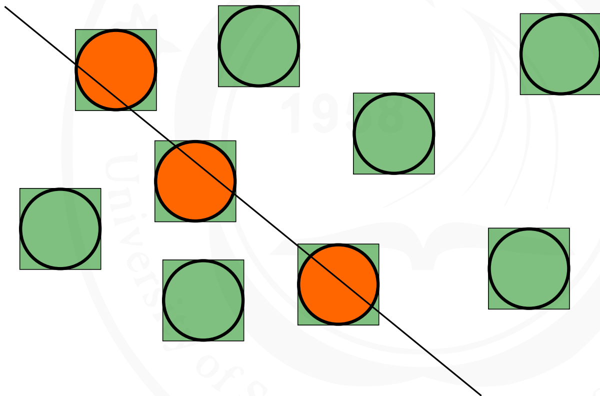
Abstracting the traversal & the container

Brute-force solution $\Rightarrow$ test all objects



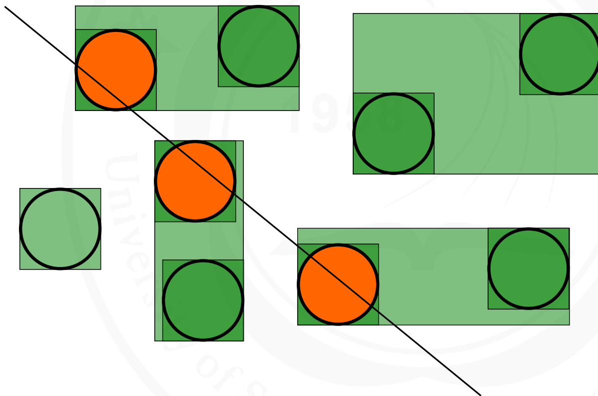
Abstracting the traversal & the container

Smart solution 1 $\Rightarrow$ Bounding Box Hierarchy



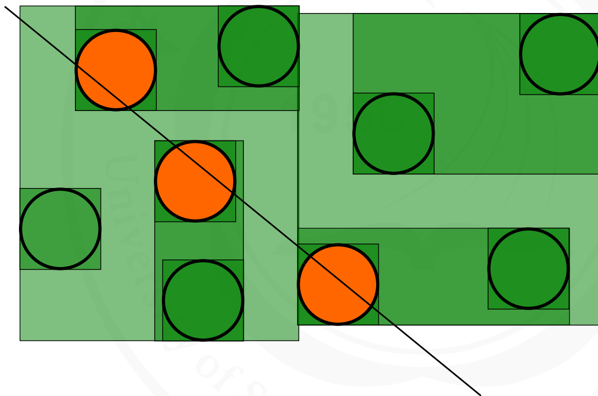
Abstracting the traversal & the container

Smart solution 1 ⇒ Bounding Box Hierarchy



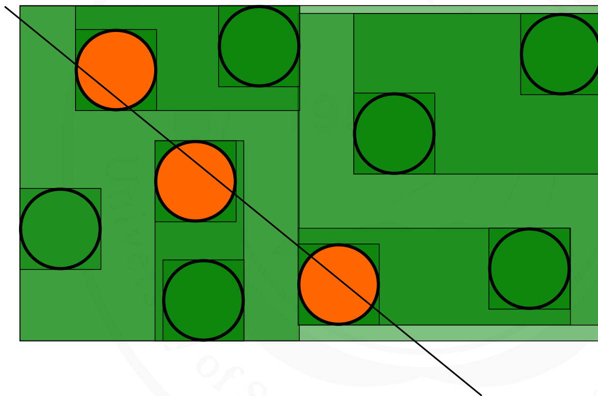
Abstracting the traversal & the container

Smart solution 1 ⇒ Bounding Box Hierarchy



Abstracting the traversal & the container

Smart solution 1 $\Rightarrow$ Bounding Box Hierarchy



Abstracting the traversal & the container

Smart solution 3 $\Rightarrow$ kd-tree

Abstracting the traversal & the container

Smart solution 4 $\Rightarrow$ Bounding Interval Hierarchy

Abstracting the traversal & the container

Each approach as an iterator of objects

1. Brute-force first ⇒ quick to get something to show
2. Bounding Box Hierarchy ⇒ large speed gain, little work
3. All the others, pick the best ⇒ competitive solution

Table of Contents

1. Introduction
2. The Iterator
3. The Visitor
4. The Factory
   - The fat constructor problem
   - The many constructors problem
   - Construction as an object
5. Conclusion

The growing interface problem

You created nice & simple objects, to handle a file system

```
interface Entity {
    Iterator<Entity> getChildren();
}
```

Listing 7: Interface of a file system entity

The growing interface problem

You created nice & simple objects, to handle a file system

```
class File implements Entity {
    Iterator<Entity> getChildren() {
      return java.utils.Collections<Entity>.EMPTY_LIST.iterator();
    }
}
```

Listing 8: Entity implementation for a file

The growing interface problem

You created nice & simple objects, to handle a file system

```
class Directory implements Entity {
  List<Entity> content;

  Iterator<Entity> getChildren() {
    return content.iterator();
  }
}
```

Listing 9: Entity implementation for a directory

The growing interface problem

Size of the content of one entity ? No problems !

```
interface Entity {
  Iterator<Entity> getChildren();

  long getContentSize();
}
```

Listing 10: Interface of a file system entity

We just add a *getContentSize* method to the interface

The growing interface problem

Size of the content of one entity ? No problems !

```
class File implements Entity {
  Iterator<Entity> getChildren() { ... }

  long getContentSize() {
    return file.getSize();
  }
}
```

Listing 11: Entity implementation for a file

We just add a *getContentSize* method to the interface

The growing interface problem

Size of the content of one entity ? No problems !

```
class Directory implements Entity {
  Iterator<Entity> getChildren() { ... }

  long getContentSize() {
    long sum = 0;
    for(Entity e : content)
      sum += e.getContentSize();
    return sum;
  }
}
```

Listing 12: Entity implementation for a directory

We just add a *getContentSize* method to the interface

The growing interface problem

Get all files with name matching a pattern ? No problems !

```
interface Entity {
  Iterator<Entity> getChildren();

  long getContentSize();

  void getMatchingFiles(String pattern, Collection<File> files);
}
```

Listing 13: Interface of a file system entity

We just add a *gatherAllFileContainingStr* method to the interface

The growing interface problem

Get all files with name matching a pattern ? No problems !

```
class File implements Entity {
  Iterator<Entity> getChildren() { ... }

  long getContentSize() { ... }

  void getMatchingFiles(String pattern, Collection<File> list) {
    if (pattern.match(file.getName()))
      list.add(this)
  }
}
```

Listing 14: Entity implementation for a file

We just add a *gatherAllFileContainingStr* method to the interface

The growing interface problem

Get all files with name matching a pattern ? No problems !

```
class Directory implements Entity {
  Iterator<Entity> getChildren() { ... }

  long getContentSize() { ... }

  void getMacthingFiles(String pattern, Collection<File> list) {
    for(Entity e : content)
      e.getMatchingFiles(pattern, list);
  }
}
```

Listing 15: Entity implementation for a directory

We just add a *gatherAllFileContainingStr* method to the interface

The growing interface problem

Pretty print the content of a hierarchy ? No problems !

```
interface Entity {
  Iterator<Entity> getChildren();

  long getContentSize();

  void getMatchingFiles(String pattern, Collection<File> files);

  void prettyPrint(OutputStream out, int padding);
}
```

Listing 16: Interface of a file system entity

We just add a *prettyPrint* method to the interface.

The growing interface problem

Pretty print the content of a hierarchy ? No problems !

```java
class File implements Entity {
  Iterator<Entity> getChildren() { ... }

  long getContentSize() { ... }

  void gatherAllFileContainingStr(String str, Collection<File> files) { ... }

  void prettyPrint(OutputStream out, int padding) {
    StringBuffer buffer = new StringBuffer();
    for(int i = 0; i < padding; ++i)
      buffer.append(' ');
    buffer.append("|-");
    buffer.append(file.getName());
    out.println(buffer.toString());
  }
}
```

Listing 17: Entity implementation for a file

We just add a *prettyPrint* method to the interface.

The growing interface problem

Pretty print the content of a hierarchy ? No problems !

```java
class Directory implements Entity {
  List<Entity> content;

  Iterator<Entity> getChildren() { ... }

  long getContentSize() { ... }

  void getMatchingFiles(String pattern, Collection<File> files) { ... }

  void prettyPrint(OutputStream out, int padding) {
    StringBuffer buffer = new StringBuffer();
    for(int i = 0; i < padding; ++i)
      buffer.append(' ');
    buffer.append("- ");
    buffer.append(dir.getName());

    out.println(buffer.toString());

    for(Entity e : content)
      e.prettyPrint(out, padding + buffer.length());
  }
}
```

Listing 18: Entity implementation for a directory

We just add a *prettyPrint* method to the interface.

The growing interface problem

The *Entity* interface keeps growing

- Mix together unrelated things
- Mix very specific with very general
- Show implementation details
- Add one feature, changes *all* implementations of *Entity*

The growing interface problem

Code growing out of control
⇒ the *bloated interface* anti-pattern

The Visitor pattern

The *Visitor* pattern allows to

- Keep the interface minimal and general
- Encapsulate algorithms working on an object hierarchy
- . . . controlled code growth !

The Visitor pattern

Our new little friend, the *EntityVisitor* interface

```
interface EntityVisitor {
  void visit(File file);

  void visit(Directory dir);
}
```

The Visitor pattern

The *Entity* interface remains simple

```
interface Entity {
  Iterator<Entity> getChildren();

  void accept(EntityVisitor visitor);
}
```

Listing 19: Interface of a file system entity

The Visitor pattern

The *File* class will not get fat

```
class File implements Entity {
  Iterator<Entity> getChildren() { ... }

  void accept(EntityVisitor visitor) {
    visitor.visit(this);
  }
}
```

Listing 20: Entity implementation for a file

The Visitor pattern

The *Directory* class will not get fat either

```
class Directory implements Entity {
  Iterator<Entity> getChildren() { ... }

  void accept(EntityVisitor visitor) {
    visitor.visit(this);
  }
}
```

Listing 21: Entity implementation for a directory

The Visitor pattern

Size of the content of one entity ? No problems !

```
class ContentSizeCalculator implements EntityVisitor {
  long getContentSize(Entity e) {
    sum = 0;
    e.accept(this);
  }

  void visit(File file) {
    sum += file.getFileHandle().getSize();
  }

  void visit(Directory dir) {
    for(Entity e : content)
      e.accept(this);
  }
}
```

ContentSizeCalculator, implementation of the *EntityVisitor* interface.

The Visitor pattern

Get all files with name matching a pattern ? No problems !

```
class FileFinder implements EntityVisitor {
  String str;
  Collection<File> list;

  void getMatchingFiles(Entity e, String pattern, Collection<File> list) {
    this.str = str;
    this.list = list;
    e.accept(this);
  }

  void visit(File file) {
    if (pattern.match(file.getName()))
      list.add(file);
  }

  void visit(Directory dir) {
    for(Entity e : content)
      e.accept(this);
  }
}
```

FileFinder, implementation of the *FileFinder* interface.

The Visitor pattern

Thanks to the *Visitor* pattern

- Adding new operations does not *modify* other code
- Each new operation is contained in one class

The Visitor pattern

Imagine the same "growing interface problem" with

- 9 different implementation of *Entity*
- 23 operations on *Entity*

The Visitor pattern

Extending the *Entity* interface

- 23×9 classes to modify
- 9 places to look for in case of bug in 1 operation

You will love your job ...

The Visitor pattern

Using the *Visitor* pattern

- 1 *EntityVisitor* interface
- 23 class for the operations
- 1 place to look for in case of bug in 1 operation

Much more convenient to maintain

Table of Contents

1. Introduction
2. The Iterator
3. The Visitor
4. The Factory
   - The fat constructor problem
   - The many constructors problem
   - Construction as an object
5. Conclusion

The fat constructor problem

You made a nice *Picture* object

```
class Picture {
  Picture(int width, int height) { ... }

  void drawLine(Point a, Point b, Color c);

  void drawPolygon(Point[] u, Color c);

  ...
}
```

The fat constructor problem

Reading *BMP* files ? No problems !

```
class Picture {
  Picture(InputStream in) {
    // BMP file decoding
  }
}
```

We just add a constructor to the class, with an input stream as parameter

The fat constructor problem

Reading *PNG* files ? No problems !

```
class Picture {
  Picture(InputStream in, PictureFileType type) {
    switch(type) {
      case BMP:
        // BMP file decoding
        break;
      case PNG:
        // PNG file decoding
        break;
    }
  }
}
```

The constructor takes an additional “picture file type” parameter

The fat constructor problem

Reading *JPEG* files ? No problems !

```
class Picture {
  Picture(InputStream in, PictureFileType type) {
    switch(type) {
      case BMP:
        // BMP file decoding
        break;
      case PNG:
        // PNG file decoding
        break;
      case JPEG:
        // JPEG file decoding
        break;
    }
  }
}
```

Just one more case to handle

The fat constructor problem

Background color for *PNG* & *BMP* files ? No problems !

```
class Picture {
  Picture(InputStream in, PictureFileType type, Color bgColor) {
    switch(type) {
      case BMP:
        // BMP file decoding, with given bg color
        break;
      case PNG:
        // PNG file decoding, with given bg color
        break;
      case JPEG:
        // JPEG file decoding
        break;
    }
  }
}
```

The constructor takes a background color parameter

The fat constructor problem

gamma color correction for *PNG* & *JPEG* files? No problems !

```
class Picture {
  Picture(InputStream in, PictureFileType type, Color bgColor, double gamma) {
    switch(type) {
      case BMP:
        // BMP file decoding, with given bg color
        break;
      case PNG:
        // PNG file decoding, with given bg color, and gamma
        break;
      case JPEG:
        // JPEG file decoding, with given and gamma
        break;
    }
  }
}
```

The constructor takes a *gamma* color correction parameter

The fat constructor problem

The *Picture* constructor keeps growing

- Complex constructor signature
- Some parameters makes sense only in some cases
- Implementation is going to be a mess
- Will grow messier with time

The fat constructor problem

Code growing out of control
⇒ the *fat constructor* anti-pattern

The many constructors problem

You made a really nice *Email* object

```
class Email {
  Email(String dst) { ... }

  void giveMeBacon() { ... }

  void makeMeCoffee() { ... }
}
```

The many constructors problem

But I want to set the receiver name sometimes ! You made a really nice *Email* object

```
class Email {
  Email(String dst) { ... }

  Email(String dst, String rcvName) { ... }
}
```

And everyone is happy now

The many constructors problem

But I want to put an attached document sometimes !

```
class Email {
  Email(String dst) { ... }

  Email(String dst, String rcvName) { ... }

  Email(String dst, File file) { ... }

  Email(String dst, String rcvName, File file) { ... }
}
```

And everyone is happy now

The many constructors problem

But I want to set the subject sometimes !

```
class Email {
  Email(String dst) { ... }

  Email(String dst, String subj) { ... }

  Email(String dst, String subj, String rcvName) { ... }

  Email(String dst, File file) { ... }

  Email(String dst, String rcvName, File file) { ... }

  Email(String dst, String subj, String rcvName, File file) { ... }
}
```

And everyone is happy now

The many constructors problem

But I want to set the *MIME* type sometimes !

```
class Email {
  Email(String dst) { ... }

  Email(String dst, String subj) { ... }

  Email(String dst, String subj, String rcvName) { ... }

  Email(String dst, File file) { ... }

  Email(String dst, String rcvName, File file) { ... }

  Email(String dst, String subj, String rcvName, File file) { ... }

  Email(String dst, MIMEType type) { ... }

  Email(String dst, String subj, MIMEType type) { ... }

  Email(String dst, String subj, String rcvName, MIMEType type) { ... }

  ...
```

Does everyone is happy yet !?

The many constructors problem

The number of constructors keeps growing

- Lot of code copy-paste very likely
- Growing interface
- Will grow messier with time

The Factory pattern

In both previous examples, we have try to "build" an object $\Rightarrow$ turn the "building" as an object itself !

The Factory pattern

A *Factory* is an interface that defines how to build a object.

The Factory pattern

The *Factory* pattern for the *Picture* example

```
class Picture {
  Picture(int width, int height) { ... }
}
```

Picture class stays clean & lean

The Factory pattern

The *Factory* pattern for the *Picture* example

```
interface PictureFactory {
  Picture getPicture();
}
```

Our new friend, the *PictureFactory* interface

The Factory pattern

The *Factory* pattern for the *Picture* example

```
class BMPPictureFactory implements PictureFactory {
  BMPFactory() { ... }

  void setInput(InputStream in) { ... }

  void setBackgroundColor(Color c) { ... }

  Picture getPicture() { ... }
}
```

Building a picture out of a *BMP* file

The Factory pattern

The *Factory* pattern for the *Picture* example

```
class PNGPictureFactory implements PictureFactory {
  PNGFactory() { ... }

  void setGamma(double gamma) { ... }

  void setInput(InputStream in) { ... }

  void setBackgroundColor(Color c) { ... }

  Picture getPicture() { ... }
}
```

Building a picture out of a *PNG* file

The Factory pattern

The *Factory* pattern for the *Picture* example

```
class JPEGPictureFactory implements PictureFactory {
  JPEGFactory() { ... }

  void setGamma(double gamma) { ... }

  void setInput(InputStream in) { ... }

  Picture getPicture() { ... }
}
```

Building a picture out of a *JPEG* file

The Factory pattern

The *Factory* pattern for the *Email* example

```
class Email {
  Email(String dst) { ... }
}
```

Email class stays clean & lean

The Factory pattern

The *Factory* pattern for the *Email* example

```
class EmailFactory {
  Email getEmail() { ... }

  void setDestination(String str) { ... }

  void setSubject(String str) { ... }

  void setMIMEType(MIMEType type) { ... }

  void setAttachement(File file) { ... }
}
```

Our new friend, the *EmailFactory* class

Table of Contents

1. Introduction
2. The Iterator
3. The Visitor
4. The Factory
   - The fat constructor problem
   - The many constructors problem
   - Construction as an object
5. Conclusion

Loose decoupling

Many design patterns work on the same ideas

- Avoiding non-controlled growth
- A distinct, clear role for each object

More patterns

There are more patterns and anti-patterns

- Read books !
- Reading forums and Q&A websites
- Reading code and API

Object programming

Good object-oriented designs achieved through *critical thinking*, not *rote learning*

- Using design patterns does not make code automatically better
- Need to carefully understand the problem