A <Basic> C++ Course

6 - The document example

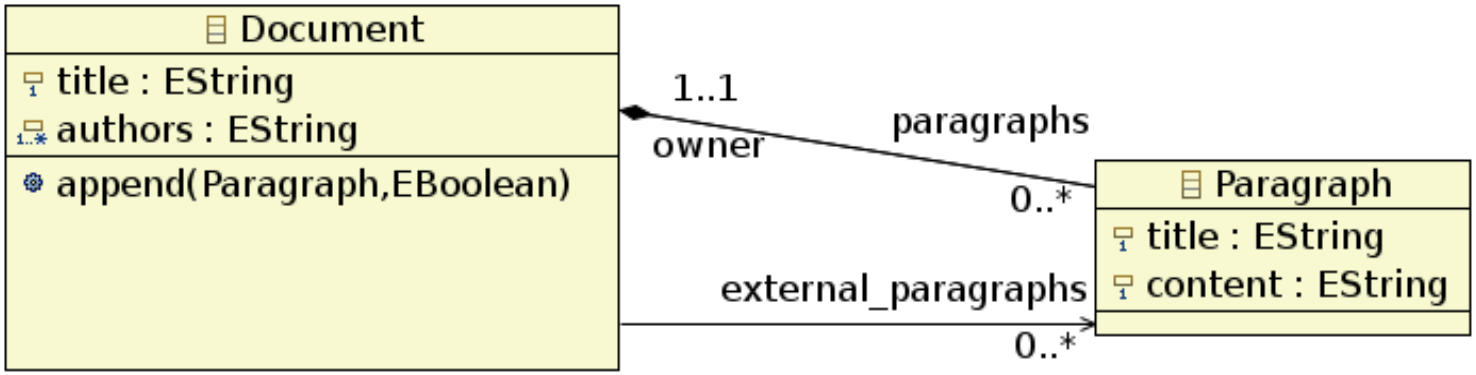
Julien Deantoni

This Week

- Just an example...

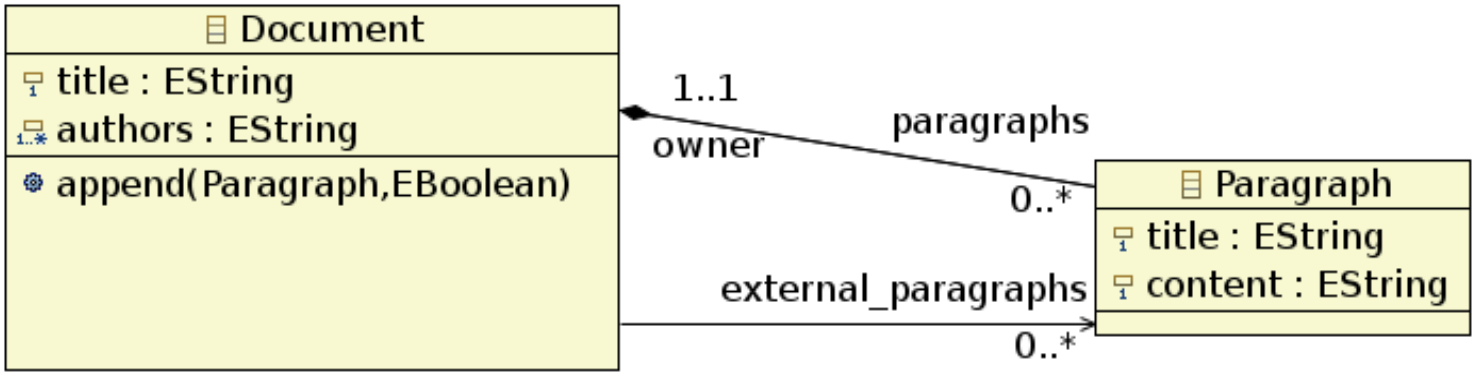
Document example: basic specifications

- Consider (unstructured) text documents. A **`Document`** is composed of:
  1. a title,
  2. A set of authors,
  3. an ordered collection of contained paragraphs,
  4. an ordered collection of referenced paragraphs.



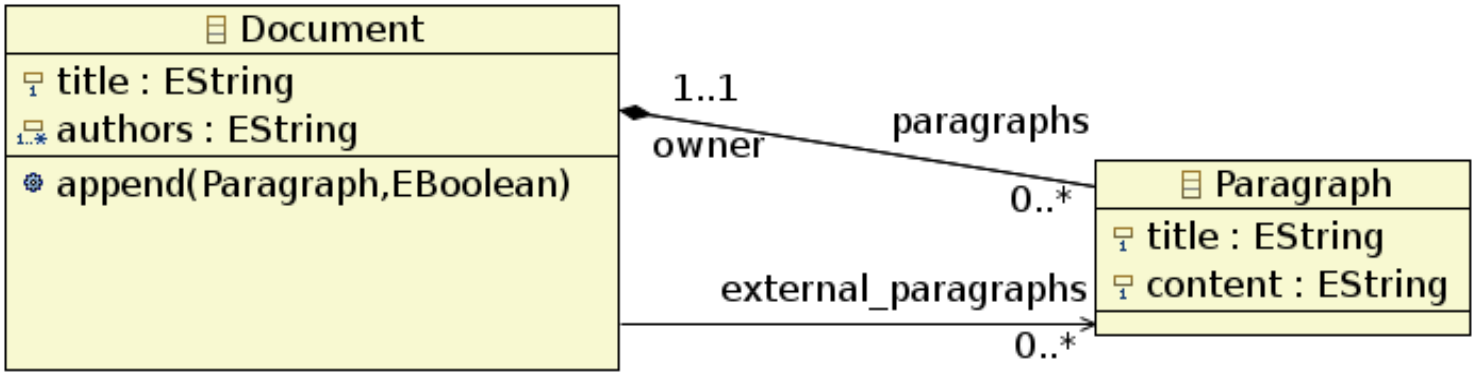
Document example:basic specifications

- Consider (unstructured) text documents. A **`Document`** offer the possibility of:
  1. Adding a paragraph. A paragraph can be add as a "`paragraphs`" or as an "`external_paragraphs`" depending on a `boolean`

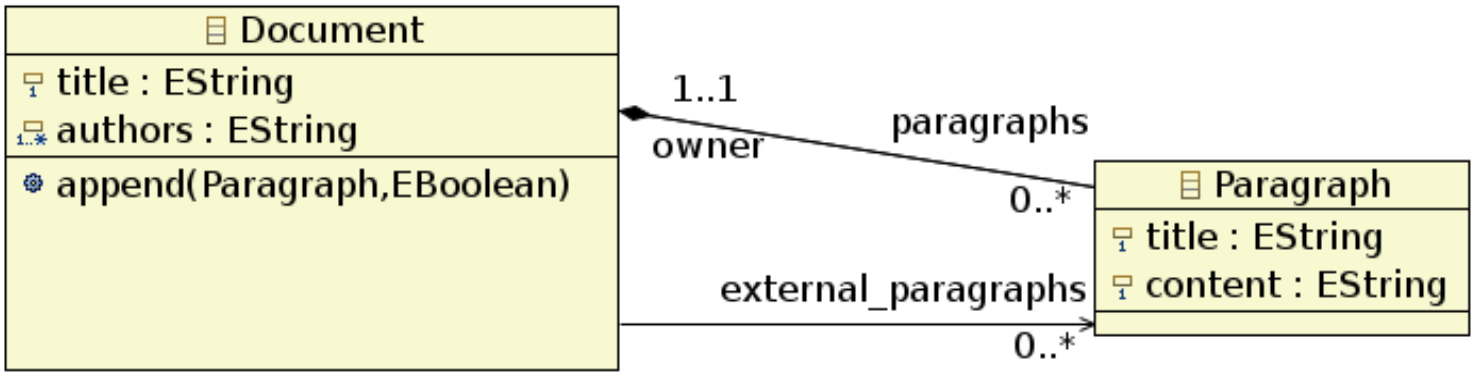


Document example:basic specifications

- A **`Paragraph`** is composed of:
  1. a title,
  2. a string representing the content,
  3. an association to its owner.



Document example: basic specifications



General title (Author, ...)

paragraph title 1

Bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla.

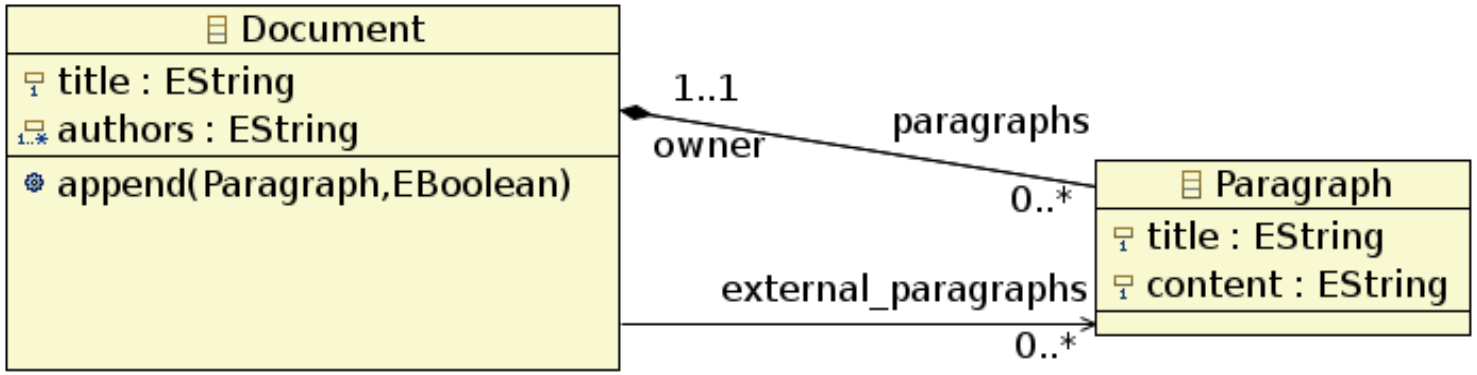
paragraph title 2

Bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla.

Document

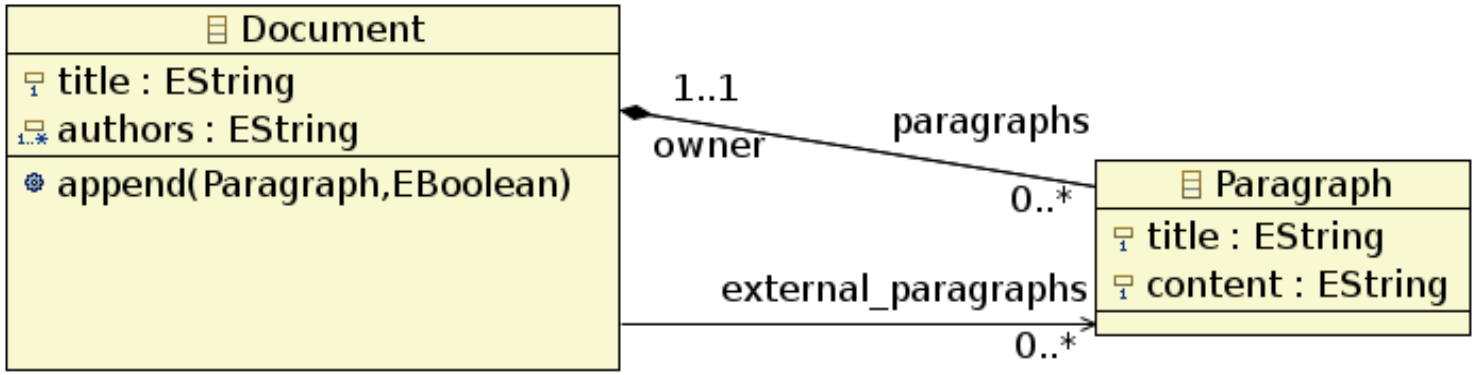
Paragraphs

Document example:basic specifications vs C++



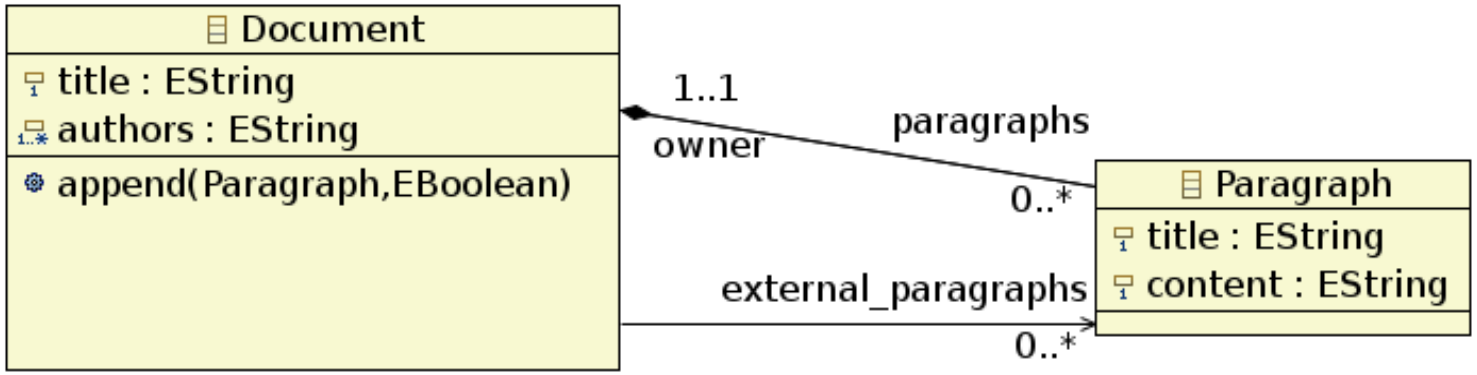

```
class Document{
private:
```

Document example:basic specifications vs C++



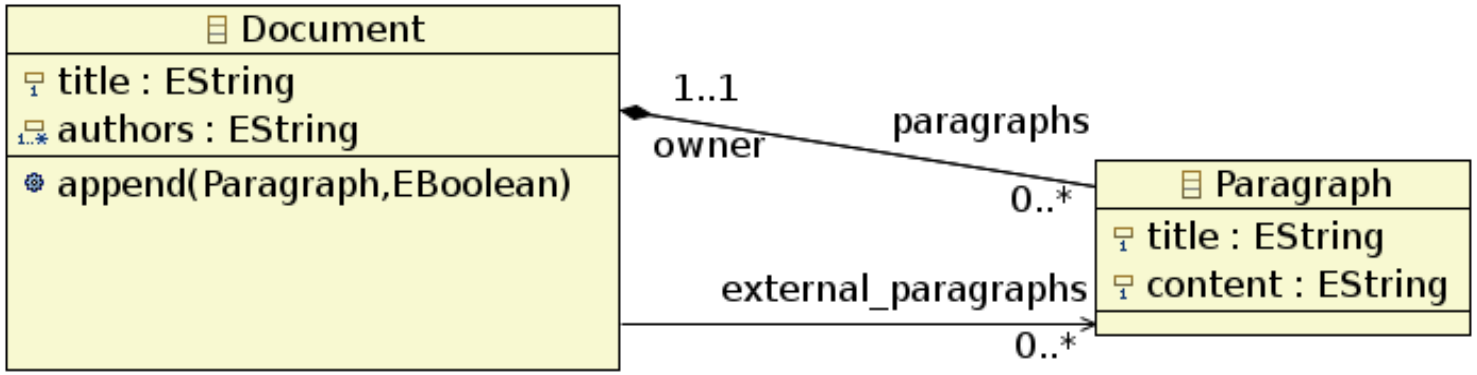

```
class Document{
private:
  string _title;
```

Document example:basic specifications vs C++



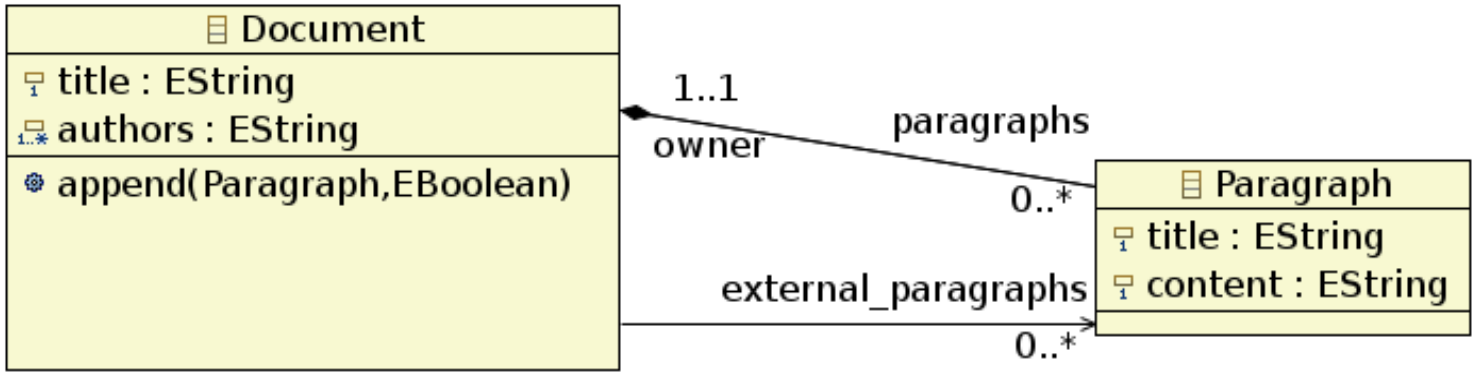

```
class Document{
private:
  string _title;
  vector<string> _authors;
```

Document example:basic specifications vs C++



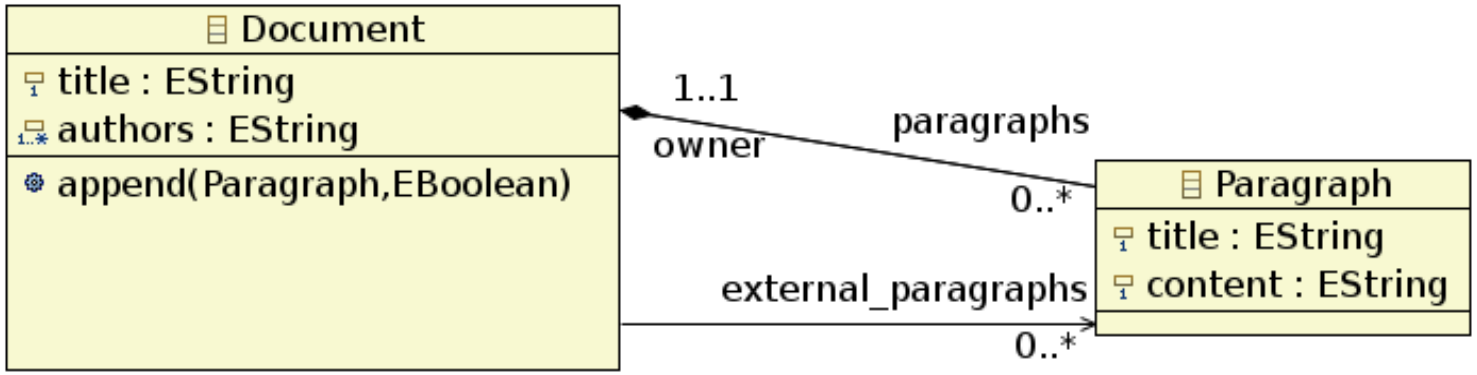

```
class Document{
private:
  string _title;
  vector<string> _authors;
  vector<Paragraph> _paragraphs;
```

Document example:basic specifications vs C++




```
class Document{
private:
  string _title;
  vector<string> _authors;
  vector<Paragraph> _paragraphs;
  vector<Paragraph*> _external_paragraphs;
```

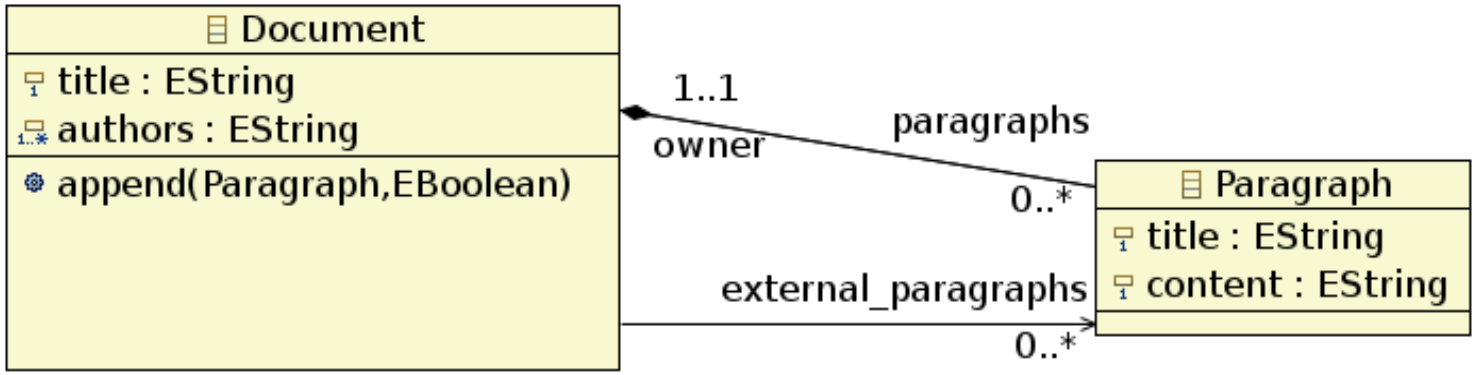
Document example:basic specifications vs C++




```
class Document{
private:
  string _title;
  vector<string> _authors;
  vector<Paragraph> _paragraphs;
  vector<Paragraph*> _external_paragraphs;
public:
```

Note the difference between containment and association

Document example:basic specifications vs C++



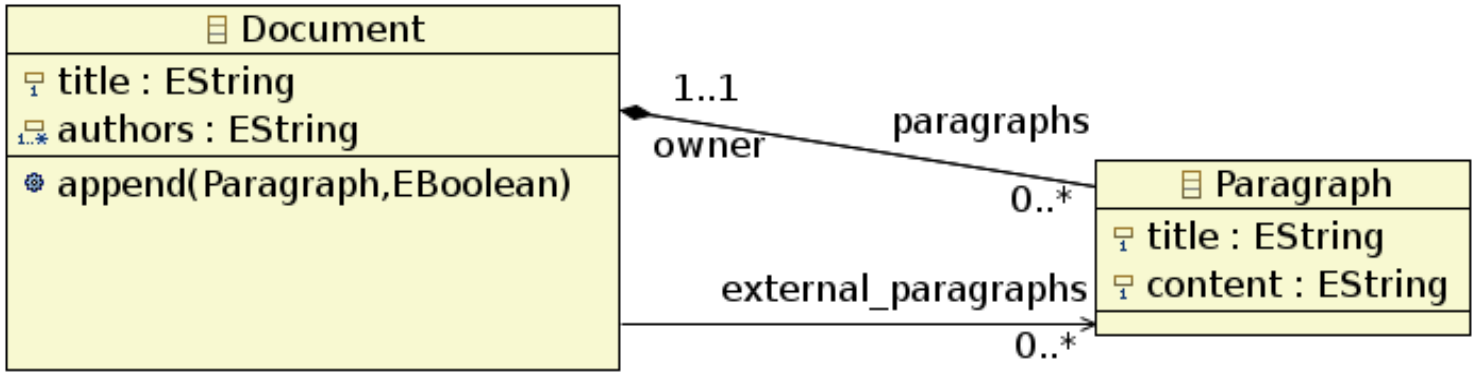

```
class Document{
private:
  string _title;
  vector<string> _authors;
  vector<Paragraph> _paragraphs;
  vector<Paragraph*> _external_paragraphs;
public:
```

Note the difference between containment and association

There are various points of view on this concern. You can choose your point of view but you will have to motivate your choice.

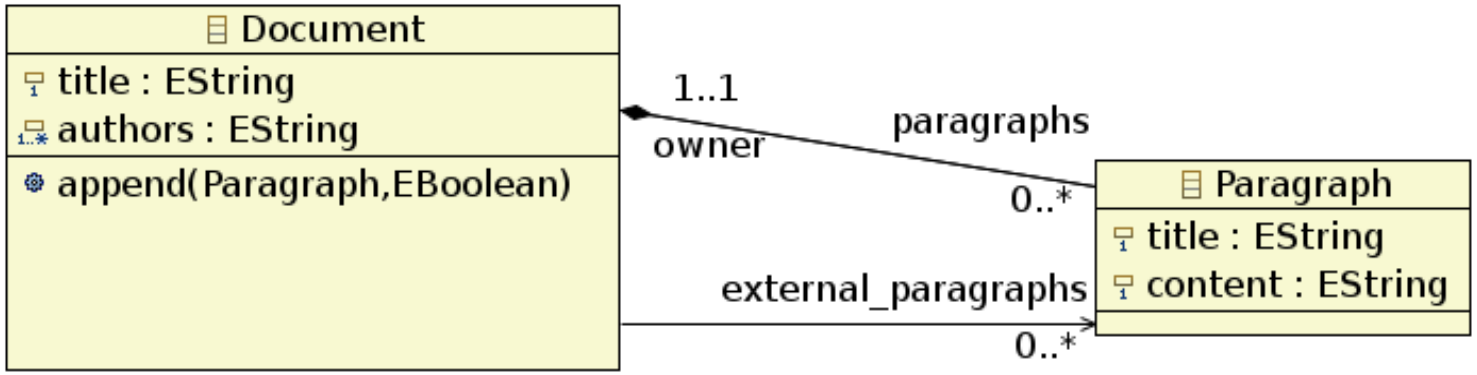
For me, when there is a containment, the object life is under the responsibility of the container. Otherwise it is not.

Document example:basic specifications vs C++




```
class Document{
private:
  string _title;
  vector<string> _authors;
  vector<Paragraph> _paragraphs;
  vector<Paragraph*> _external_paragraphs;
public:
  /*! constructors */
  Document(string title="default_title",
        vector<string> new_authors= vector<string>(),
         vector<Paragraph> new_paragraphs=vector<Paragraph>(),
          vector<Paragraph*> new_external_paragraphs=vector<Paragraph*>());
```

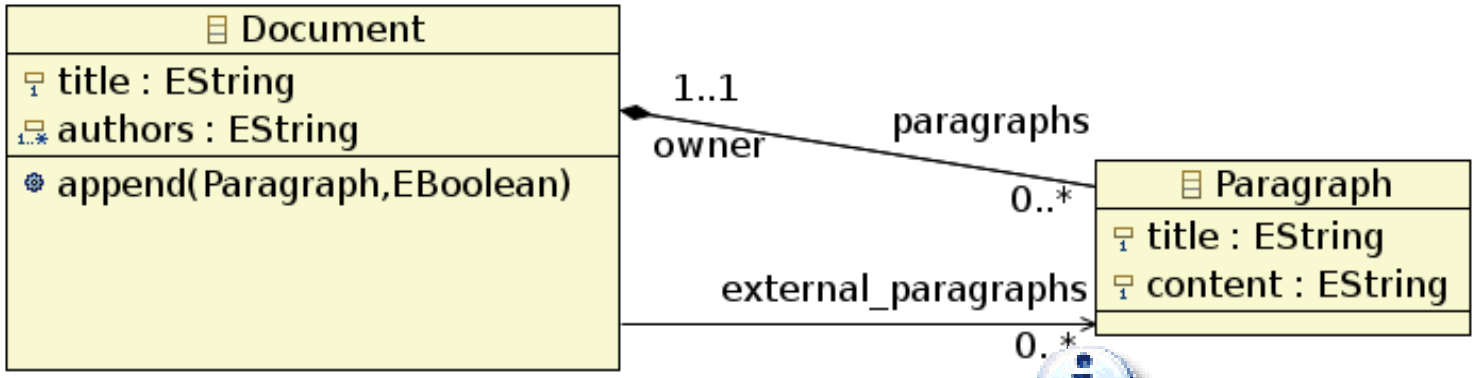
Document example:basic specifications vs C++




```
class Document{
private:
  string _title;
  vector<string> _authors;
  vector<Paragraph> _paragraphs;
  vector<Paragraph*> _external_paragraphs;
public:
  /*! constructors */
  Document(string title="default_title",
        vector<string> new_authors= vector<string>(),
         vector<Paragraph> new_paragraphs=vector<Paragraph>(),
          vector<Paragraph*> new_external_paragraphs=vector<Paragraph*>());
```

Default parameters that explicitly call a constructor

Document example:basic specifications vs C++

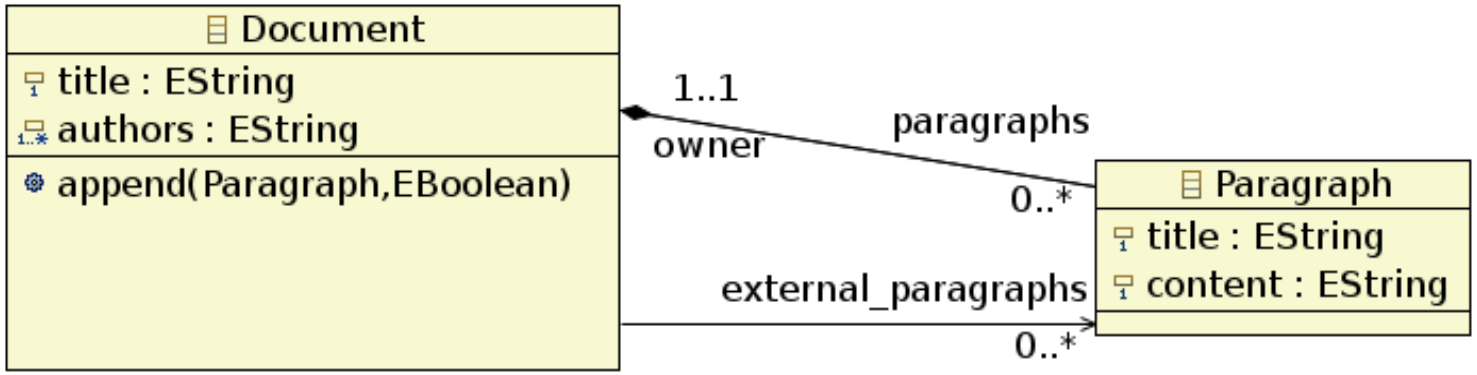



```
class Document{
private:
  string _title;
  vector<string> _authors;
  vector<Paragraph> _paragraphs;
  vector<Paragraph*> _external_paragraphs;
public:
  /*! constructors */
  Document(string title="default_title",
        vector<string> new_authors= vector<string>(),
         vector<Paragraph> new_paragraphs=vector<Paragraph>(),
          vector<Paragraph*> new_external_paragraphs=vector<Paragraph*>());
```

A name must be set to explicitly call a constructor

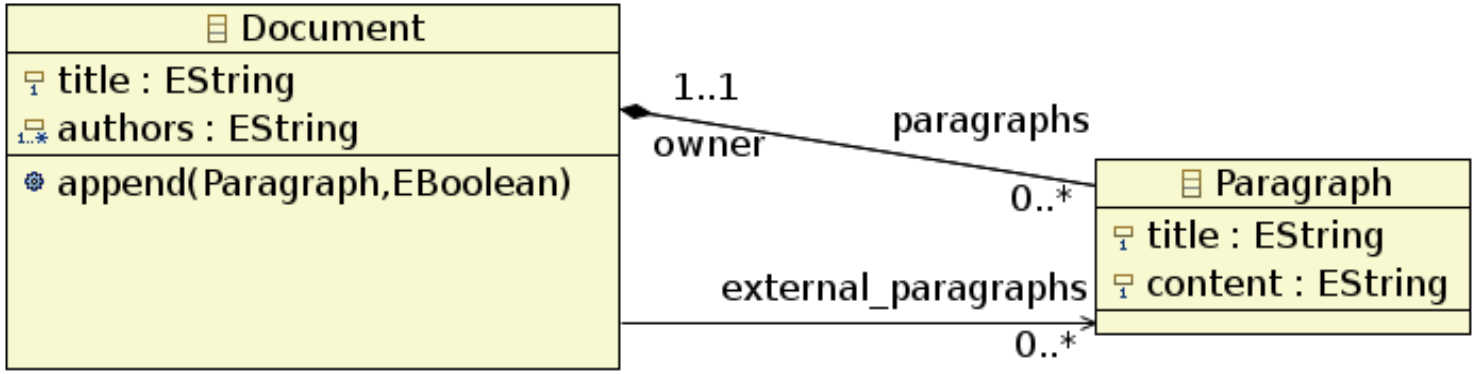
Default parameters that explicitly call a constructor

Document example:basic specifications vs C++




```
class Document{
private:
  string _title;
  vector<string> _authors;
  vector<Paragraph> _paragraphs;
  vector<Paragraph*> _external_paragraphs;
public:
  /*! constructors */
  Document(string title="default_title",
        vector<string> new_authors= {},
         vector<Paragraph> new_paragraphs={},
          vector<Paragraph*> new_external_paragraphs={});
```

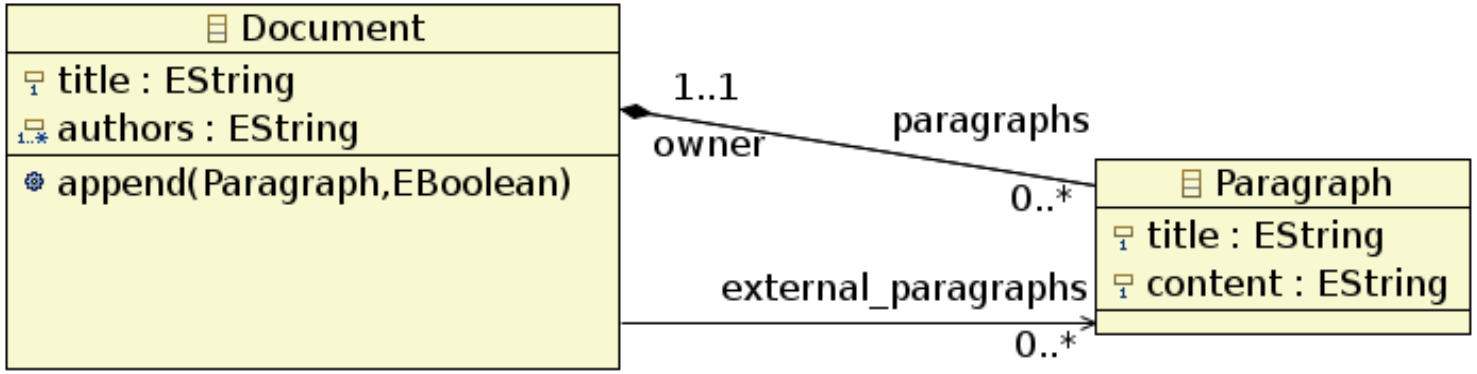
Document example:basic specifications vs C++




```
class Document{
private:
  string _title;
  vector<string> _authors;
  vector<Paragraph> _paragraphs;
  vector<Paragraph*> _external_paragraphs;
public:
  /*! constructors */
  Document(string title="default_title",
        vector<string> new_authors= vector<string>(),
         vector<Paragraph> new_paragraphs=vector<Paragraph>(),
          vector<Paragraph*> new_external_paragraphs=vector<Paragraph*>());
  //! Copy constructor
  Document(const Document&);
```

Should we use the default one or not ?

Document example:basic specifications vs C++

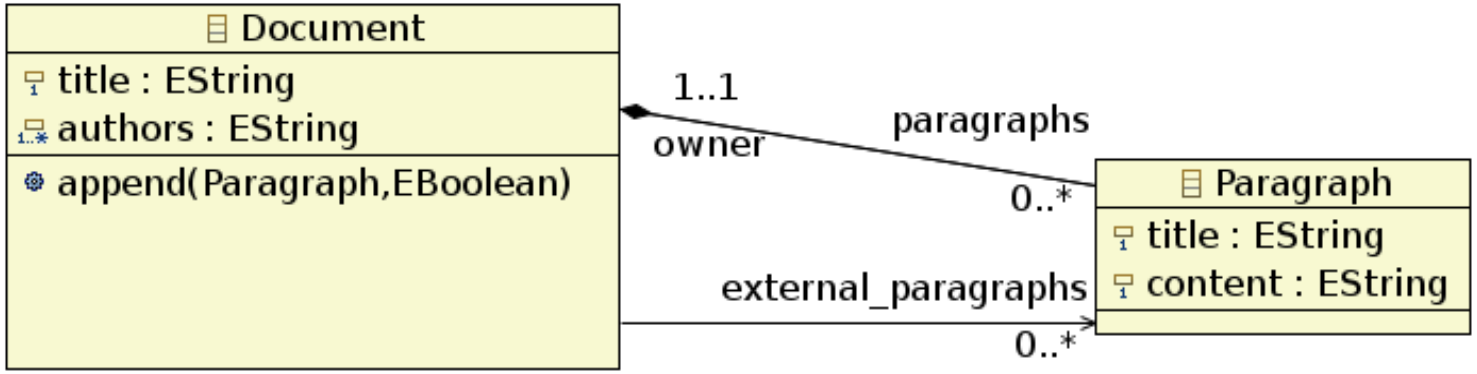



```cpp
/*!
* We cannot use the default copy ctor since we have to set the owner of the contained paragraphs.
*/
Document::Document(const Document& d)
:_title(d._title), _authors(d._authors), _paragraphs(d._paragraphs), _external_paragraphs(d._external_paragraphs)
{
  for (Paragraph& p :_paragraphs){
      p.set_owner( &*this);
  }
}
```

```
        vector<Paragraph> new_paragraphs=vector<Paragraph>(),
         vector<Paragraph*> new_external_paragraphs=vector<Paragraph*>());
  //! Copy constructor
  Document(const Document&);
```

*Should we use the default one or not ? → **NO***

Document example:basic specifications vs C++

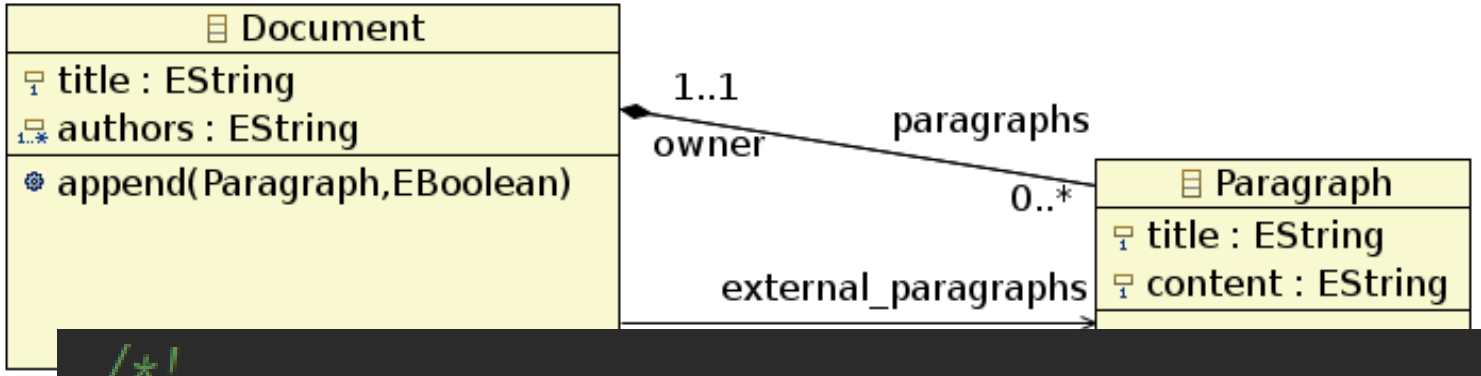



```
class Do
private:
  string
  vector
  vector
  vector
public:
  /*! co
  Docume


  //! Co
  Docume
```

```
/*!
* We cannot use the default copy ctor since we have to set the owner
*/
Document::Document(const Document& d)
:_title(d._title), _authors(d._authors), _paragraphs(d._paragraphs)
{
    for (Paragraph& p :_paragraphs){
        p.set_owner( &:*this);
    }
}
```

What if I'm not using a reference ?

Document example:basic specifications vs C++

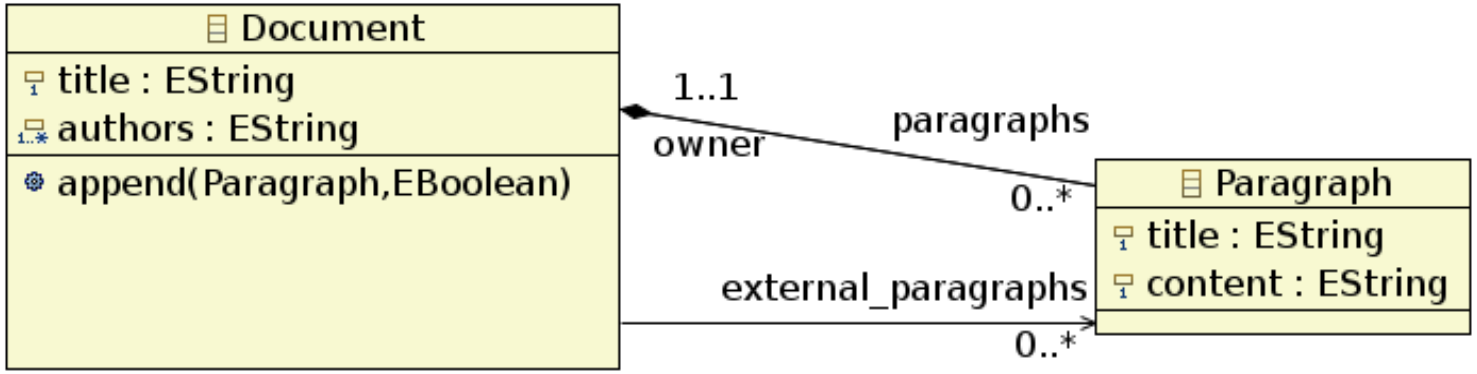



```
class D
private
  strin
  vecto
  vecto
  vecto
public:
  /*! c
  Docum

  //! c
  Docum
```

```
/*!
* We cannot use the default copy ctor since we have to set th
*/
Document::Document(const Document& d)
:_title(d._title), _authors(d._authors), _paragraphs(d._para
{
    for (Paragraph p :_paragraphs){
        p.set_owner( &: *this);
    }
}
```

What if I'm not using a reference ?

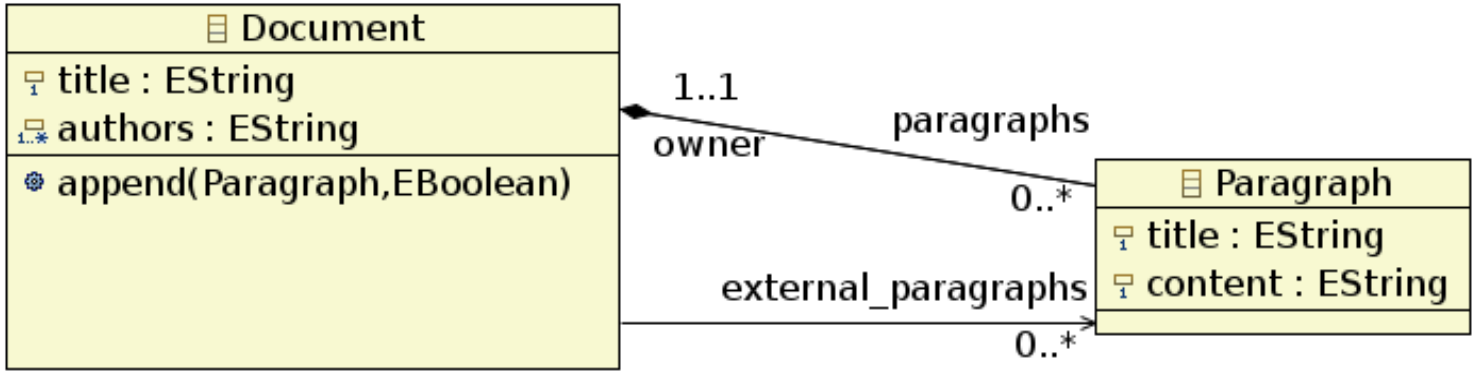
Document example:basic specifications vs C++




```
class Document{
private:
  string _title;
  vector<string> _authors;
  vector<Paragraph> _paragraphs;
  vector<Paragraph*> _external_paragraphs;
public:
  /*! constructors */
  Document(string title="default_title",
         vector<string> new_authors= vector<string>()
          vector<Paragraph> new_paragraphs=vector<Paragraph>(),
           vector<Paragraph*> new_external_paragraphs=vector<Paragraph*>());
  //! Copy constructor
  Document(const Document&);
  + all the relevant accessors !! (get / set / add / remove)
```

Accessors allow control

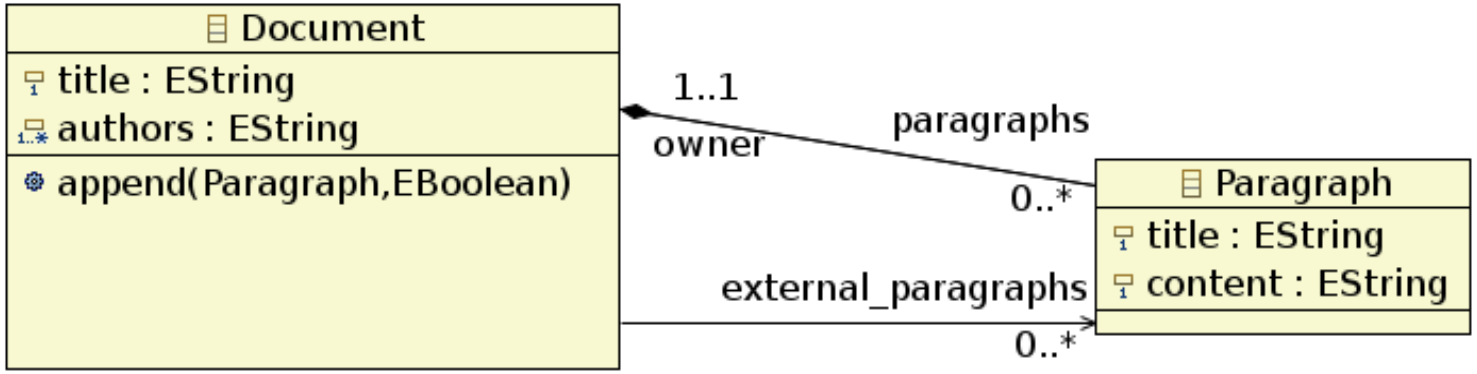
Document example:basic specifications vs C++



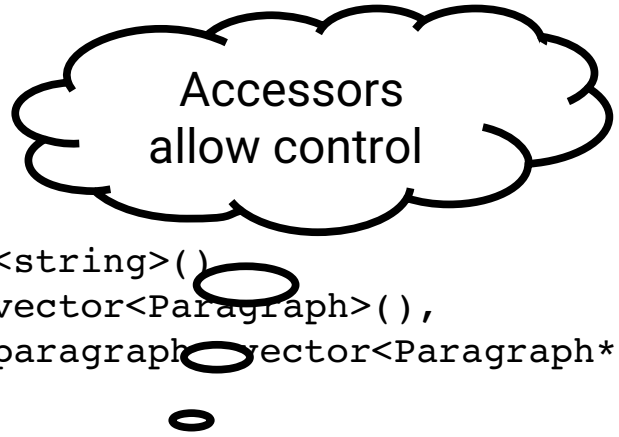

```
class Document{
private:
  string _title;
  vector<string> _authors;
  vector<Paragraph> _paragraphs;
  vector<Paragraph*> _external_paragraphs;
public:
  /*! constructors */
  Document(string title="default_title",
         vector<string> new_authors= vector<string>()
          vector<Paragraph> new_paragraphs=vector<Paragraph>(),
           vector<Paragraph*> new_external_paragraphs=vector<Paragraph*>());
  //! Copy constructor
  Document(const Document&);
  + all the relevant accessors !! (get / set / add / remove)
  void append
};
```

Accessors allow control

Document example:basic specifications vs C++




```
class Document{
private:
  string _title;
  vector<string> _authors;
  vector<Paragraph> _paragraphs;
  vector<Paragraph*> _external_paragraphs;
public:
  /*! constructors */
  Document(string title="default_title",
         vector<string> new_authors= vector<string>()
          vector<Paragraph> new_paragraphs=vector<Paragraph>(),
           vector<Paragraph*> new_external_paragraphs=vector<Paragraph*>());
  //! Copy constructor
  Document(const Document&);
  + all the relevant accessors !! (get / set / add / remove)
  void append(Paragraph&, bool);
};
```



Document example:basic specifications vs C++

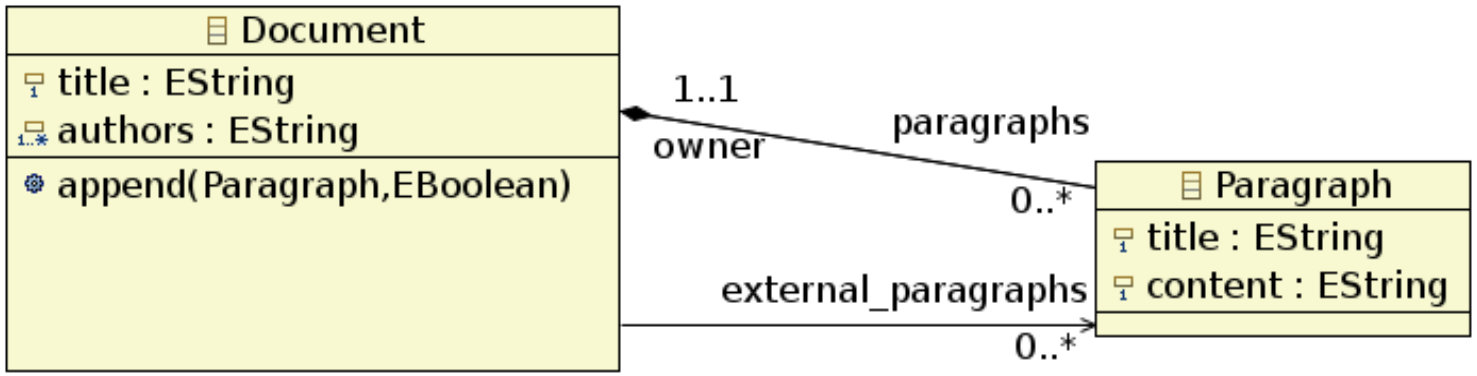
```
#ifndef _DOCUMENT_H_
#define _DOCUMENT_H_
#include <string>
#include <vector>
#include "paragraph.h"
using std::string;
using std::vector;

class Document{
private:
  string _title;
  vector<string> _authors;
  vector<Paragraph> _paragraphs;
  vector<Paragraph*> _external_paragraphs;
public:
  //! constructors with default parameters
  Document(string title="default_title",
         vector<string> new_authors= vector<string>(),
          vector<Paragraph> new_paragraphs=vector<Paragraph>(),
           vector<Paragraph*> new_external_paragraphs=vector<Paragraph*>());
  //! Copy constructor
  Document(const Document&);
  // +all the relevant accessors !! (get / set)
  void append(Paragraph&, bool);
};
#endif
```

Document example:basic specifications vs C++

```
#include "document.h"


Document::Document(string title,
                   vector<string> new_authors,
                   vector<Paragraph> new_paragraphs,
                   vector<Paragraph*> new_external_paragraphs)
: title(title),
 _authors(new_authors),
 _paragraphs(new_paragraphs),
 _external_paragraphs(new_external_paragraphs)
{
  for (Paragraph& p :_paragraphs){
      p.set_owner(*this);
  }
}

Document::Document(const Document& d)
: title(d.title),
 _authors(d.new_authors),
 _paragraphs(d.new_paragraphs),
 _external_paragraphs(d.new_external_paragraphs)
{
 for (Paragraph& p :_paragraphs){
      p.set_owner(*this);
  }
}
```

Document example:basic specifications vs C++

```
#include "document.h"
```

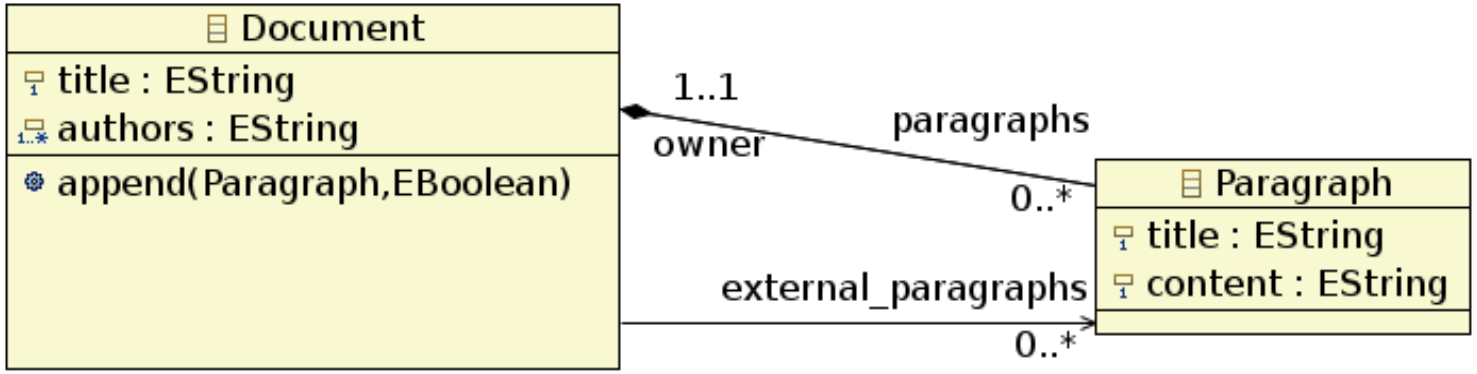



```
void Document::append(Paragraph& p, bool owned)
{
  if (owned == true)
  {
    p.owner = this;
    _paragraphs.push_back(
  }
  else
  {
    _external_paragraphs.push_back(
  }
}
```

document.cpp

Document example:basic specifications vs C++

```
#include "document.h"
```

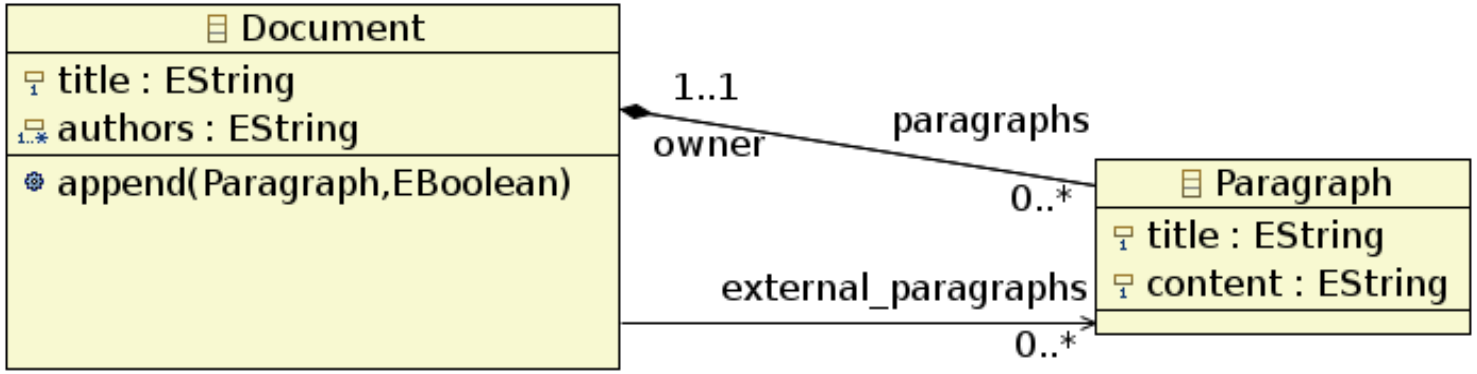



```
void Document::append(Paragraph& p, bool owned)
{
  if (owned == true)
  {
    p.owner = this;
    _paragraphs.push_back(p);
  }
  else
  {
    _external_paragraphs.push_back(
  }
}
```

document.cpp

Document example:basic specifications vs C++

```
#include "document.h"
```




```
void Document::append(Paragraph& p, bool owned)
{
  if (owned == true)
  {
    p.owner = this;
    _paragraphs.push_back(p);
  }
  else
  {
    _external_paragraphs.push_back(&p);
  }
}
```

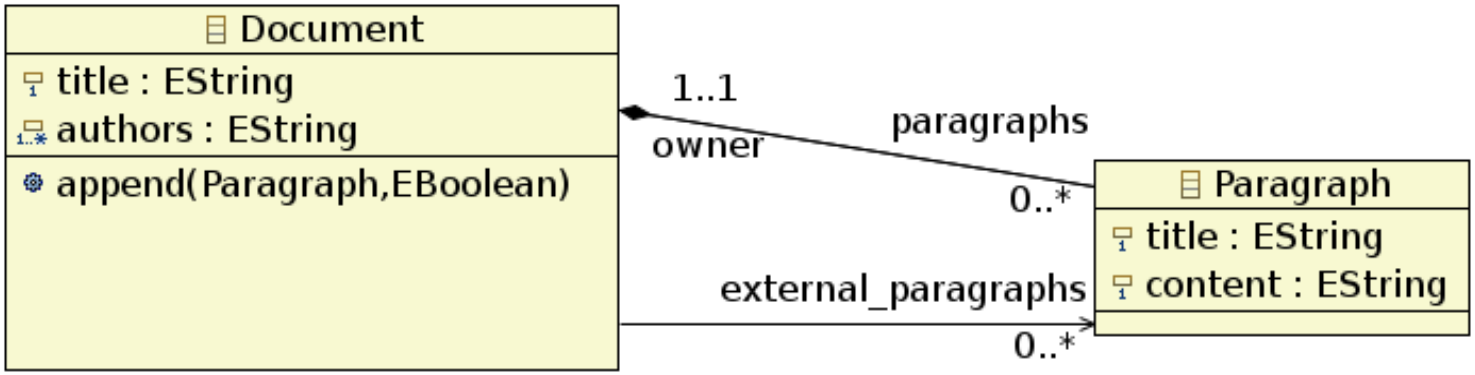
document.cpp

Document example:basic specifications vs C++

```
#include "document.h"
/*!
 * Three last parameters have a default value: empty vector.
 */
Document::Document(string title,
                   vector<string> new_authors,
                   vector<Paragraph> new_paragraphs,
                   vector<Paragraph*> new_external_paragraphs)
: title(title),
 _authors(new_authors),
 _paragraphs(new_paragraphs),
 _external_paragraphs(new_external_paragraphs)
{
 for (Paragraph& p :_paragraphs){p.set_owner(*this); }
}
/*! append add a paragraph to 'this' document.
 *    \param p Paragraph: the paragraph to add to the document
 *    \param owned boolean: if true, the paragraph is add to the document
 *           as internal resource, otherwise, as external resource
 */
void Document::append(Paragraph& p, bool owned)
{
  if (owned == true)
  { _paragraphs.push_back(Paragraph(p));}
  else
  { _external_paragraphs.push_back(&p);}
}
```

document.cpp

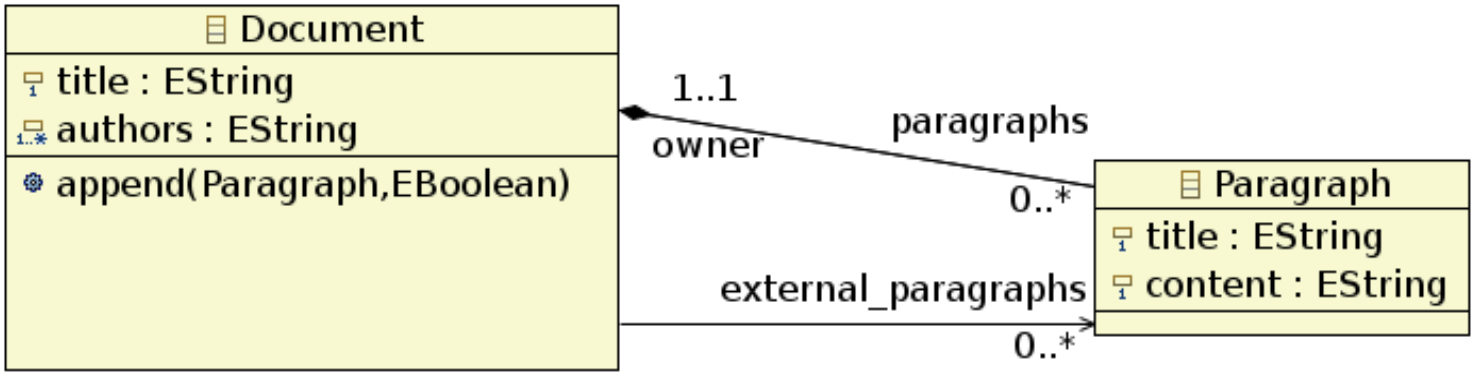
Document example:basic specifications vs C++




```
class Paragraph{
private:
  string _title;
  string _content;
  Document* _owner;
  ...
```

paragraph.h

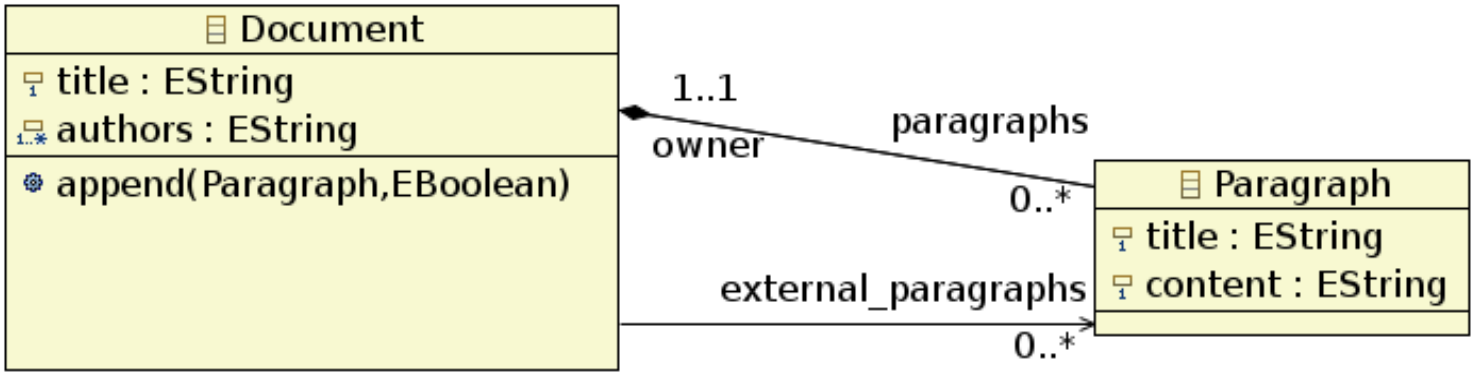
Document example:basic specifications vs C++




```
class Paragraph{
private:
  string _title;
  string _content;
  Document* _owner = nullptr;
public:
  /*! constructors */
  Paragraph(const Paragraph&) = default;
  Paragraph(string t="default_title", string c="");
  Paragraph(string, string, Document&);

//...
};
```

paragraph.h

Document example:basic specifications vs C++




```
class Paragraph{
private:
  string _title;
  string _content;
  Document* _owner = nullptr;
public:
  /*! constructors */
  Paragraph(const Paragraph&) = default;
  Paragraph(string t="default_title", string c="");
  Paragraph(string, string, Document&);

//...
};
```

Should we forbid construction of Paragraph with no owner ?

paragraph.h

Document example:basic specifications vs C++

```
#ifndef _PARAGRAPH_H_
#define _PARAGRAPH_H_
#include <string>
#include <vector>

class Document;

class Paragraph{
private:
  string _title;
  string _content;
  Document* _owner = nullptr;
public:
  /*! constructors */
  Paragraph(const Paragraph&) = default;
  Paragraph(string t="default_title", string c=""); //no owner to ease testing phase
  Paragraph(Document&, string t="default_title", string c="");

//...
};

#endif
```

paragraph.h

Document example:basic specifications vs C++

```
#ifndef _PARAGRAPH_H_
#define _PARAGRAPH_H_
#include <string>
#include <vector>

class Document;

class Paragraph{
private:
  string _title;
  string _content;
  Document* _owner;
public:
  /*! constructors */
  Paragraph(const Paragraph&) = default;
  Paragraph(string t="default_title", string c=""); //no owner to ease testing phase
  Paragraph(Document&, string t="default_title", string c="");

//...
};

#endif
```

Just says to the compiler that Document exists elsewhere

paragraph.h

Document example:basic specifications vs C++

```
#include "paragraph.h"
Paragraph::Paragraph(const Paragraph& p)
:_title(p._title),
 _content(p._content),
 _owner(p._owner)
{}

Paragraph::Paragraph(string title, string content)
:_title(title),
 _content(content),
 _owner(nullptr)
{}

Paragraph::Paragraph(Document& owner, string title, string content)
:_title(title),
 _content(content),
 _owner(&owner)
{}
```

Document example:basic specifications vs C++

```
#include "paragraph.h"
Paragraph::Paragraph(const Paragraph& p)
:_title(p._title),
 _content(p._content),
 _owner(p._owner)
{}

Paragraph::Paragraph(string title, string content)
:_title(title),
 _content(content),
 _owner(nullptr)
{}

Paragraph::Paragraph(Document& owner, string title, string content)
:_title(title),
 _content(content),
 _owner(&owner)
{}

Paragraph::Paragraph(Document& owner, string title, string content)
:Paragraph(title, content)
{
_owner = &owner;
}
```

Using constructor delegation //c++11

Document example:basic specifications vs C++

```
#include "paragraph.h"
Paragraph::Paragraph(const Paragraph& p)
:_title(p._title),
 _content(p._content),
 _owner(p._owner)
{}

Paragraph::Paragraph(s
:_title(title),
 _content(content),
 _owner(nullptr)
{}

Paragraph::Paragraph(Document& owner, string title, string content)
:_title(title),
 _content(content),
 _owner(&owner)
{}

Paragraph::Paragraph(Document& owner, string title, string content)
:Paragraph(title, content)
{
_owner = &owner;
}
```

Different use of the '&'

Using constructor delegation //c++11

Document example:basic specifications vs C++

```
#include "document.h"
int main(int argc, char* argv[])
{
 /*paragraph creation*/
  Paragraph p1{"Un titre","des trucs à lire ... \n\t\t blablabla "};
  Paragraph p2{"Un autre titre","des trucs autre trucs"};
  Paragraph pext1{"Un titre externe","un paragraph externe"};

 /*author vector creation*/
  vector<string> d1_authors = {"Julien Deantoni"};
  d1_authors.push_back("Jean-Paul Rigault");

 /*paragraph vector creation*/
  vector<Paragraph> d1_paragraphs = {p1};

 /*document creation*/
  Document d1{"Nouveau Document", d1_authors, d1_paragraphs};

 /*add of an internal paragraph*/
  d1.append(p2, true);

 /*add of an external paragraph*/
  d1.append(pext1, false);
return 0;
}
```

//C++11 initializations

main.cpp

Document example:basic specifications vs C++

```
#include "document.h"                                   //C++11 initializations
int main(int argc, char* argv[])
{
 /*paragraph creation*/
  Paragraph p1 ={"Un titre","des trucs à lire ... \n\t\t blablabla "};
  Paragraph p2 = {"Un autre titre","des trucs autre trucs"};
  Paragraph pext1 = {"Un titre externe","un paragraph externe"};

 /*author vector creation*/
  vector<string> d1_authors ={"Julien Deantoni"};
  d1_authors.push_back("Jean-Paul Rigault");

 /*paragraph vector creation*/
  vector<Paragraph> d1_paragraphs={p1};

 /*document creation*/
  Document d1 ={"Nouveau Document", d1_authors, d1_paragraphs};

Document d1bis={"Nouveau Document", {"Julien Deantoni","Jean-Paul Rigault"}, {p1, p2}};

 /*add of an internal paragraph*/
  d1.append(p2, true);

 /*add of an external paragraph*/
  d1.append(pext1, false);
return 0;
}
```

main.cpp

Document example:basic specifications vs C++

```
#include "document.h"                                   //C++11 initializations
int main(int argc, char* argv[])
{
 /*paragraph creation*/
  Paragraph p1 ={"Un titre","des trucs à lire ... \n\t\t blablabla "};
  Paragraph p2 = {"Un autre titre","des trucs autre trucs"};
  Paragraph pext1 = {"Un titre externe","un paragraph externe"};

 /*author vector creation*/
  vector<string> d1_authors ={"Julien Deantoni"};
  d1_authors.push_back("Jean-Paul Rigault");

 /*paragraph vector creation*/
  vector<Paragraph> d1_paragraphs={p1};

 /*document creation*/
  Document d1 ={"Nouveau Document", d1_authors, d1_paragraphs};

Document d1bis={"Nouveau Document", {"Julien Deantoni","Jean-Paul Rigault"}, {p1}};

 /*add of an internal paragraph*/
  d1.append(p2, true);

 /*add of an external paragraph*/
  d1.append(pext1, false);
return 0;
}
```

Without a printing function it is hard to know if it works fine

main.cpp

Document example: operator overloading

```
#ifndef _DOCUMENT_H_
#define _DOCUMENT_H_

#include <string>
#include <vector>
#include <iostream>
#include "paragraph.h"

class Document{
private:
  string _title;
  vector<string> _authors;
  vector<Paragraph> _paragraphs;
  vector<Paragraph*> _external_paragraphs;
public:
  //...

  friend ostream& operator<<(ostream&, const Document&);
};

#endif
```

document.h

Document example: operator overloading

```
//C++11 loops
#include "document.h"

//....


  std::ostream& operator<<(std::ostream& os, const Document& d)
  {
    os << d._title << " (";

    for(const std::string author : d._authors){
        os << author << ' ';
    }
    os << ")\n";
    for (Paragraph& p : d._paragraphs){
         os << p;
    }
    for (Paragraph* ptr_ep : d._external_paragraphs){
         os << *(ptr_ep);
    }
    return os;
  }
```

document.cpp

Document example: operator overloading

```
                                                          //C++11 loops
#include "document.h"

//....


  std::ostream& operator<<(std::ostream& os, const Document& d)
  {
    os << d._title << " (";

    for(const                                              {
        os <<
    }
    os << ")\
    for (Parag
        os << p;
    }
    for (Paragraph* ptr_ep : d._external_paragraphs){
        os << *(ptr_ep);
    }
    return os;
  }
```

Dereference the pointer because we want to print the paragraph, not the pointer

document.cpp

Document example: operator overloading

```
                                                        //C++11 loops
#include "document.h"

//....


  std::ostream& operator<<(std::ostream& os, const Document& d)
  {
    os << d._title << " (";

    for(const std::string& author : d._authors){
        os << author << ' ';
    }
    os << ")\n";
    for (Paragraph& p : d._paragraphs){
         os << p;
    }
    for (Paragraph* ptr_ep : d._external_paragraphs){
         if (ptr_ep != nullptr) {os << *(ptr_ep);}
    }
    return os;
  }
```

document.cpp

Document example: operator overloading

```
//C++11 loops
#include "document.h"

//....


  std::ostream& oper                                    nst Document& d)
  {
    os << d._title

    for(const std::str                                  hors){
        os << autho          ' ';
    }
    os << ")\n";
    for (Paragraph& p : d._paragraphs){
         os << p;
    }
    for (Paragraph* ptr_ep : d._exter al_paragraphs){
         if (ptr_ep != nullptr) {os << *(ptr_ep);}
    }
    return os;
  }
```

Use of the **operator<<** from the Paragraph class

Use of the **operator<<** from the Paragraph class

Document example: operator overloading

```
#ifndef _PARAGRAPH_H_
#define _PARAGRAPH_H_
#include <string>
#include <vector>
#include <iostream>

class Document;

class Paragraph{
private:
  string _title;
  string _content;
  Document* _owner;
public:
        //...


  friend ostream& operator<<(ostream&, const Paragraph&);

};

#endif
```

paragraph.h

Document example: operator overloading

```
#include "paragraph.h"



//....




  ostream& operator<<(ostream& os, const Paragraph& p)
  {
    os<< '\t' << p._title << “\n” << "\t\t" << p._content << endl;
    return os;
  }
```

paragraph.cpp

Document example: operator overloading

```
#include "document.h"
int main(int argc, char* argv[])
{
 /*paragraph creation*/
  Paragraph p1={"Un titre","des trucs à lire ... \n\t\t blablabla "};
  Paragraph p2={"Un autre titre","des trucs autre trucs"};
  Paragraph pext1={"Un titre externe","un paragraph externe"};

 /*author vector creation*/
  vector<string> d1_authors={"Julien DeAntoni","Jean-Paul Rigault"};

 /*paragraph vector creation*/
  vector<Paragraph> d1_paragraphs={p1};

 /*document creation*/
  Document d1("Nouveau Document", d1_authors, d1_paragraphs);

 /*add of an internal paragraph*/
  d1.append(p2, true);

 /*add of an external paragraph*/
  d1.append(pext1, false);

 cout<<"**************************************************"<<endl;
 cout<< d1 << endl;

return 1;
}
```

main.cpp

Document example: operator overloading

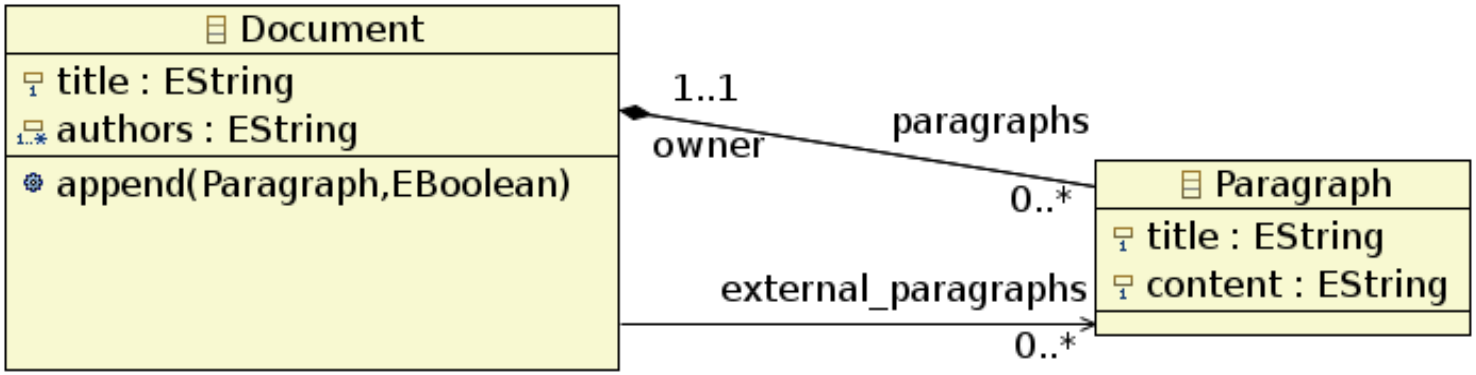
```
#include "document.h"
int main(int argc, char* argv[])
{
 /*para
  Paragr
  Paragr
  Paragr

 /*autho
  vector
  d1_aut
  d1_aut

 /*para
  vector
  d1_par

 /*docur
  Docume

 /*add
  d1.app

 /*add
  d1.app

 cout<<"*****************************************************"<<endl;
 cout<< d1 << endl;

return 1;
}
```

```
*****************************************************
Nouveau Document (Julien DeAntoni Jean-Paul Rigault)
        Un titre
                des trucs à lire ...
                 blablabla
        Un autre titre
                des trucs autre trucs
        Un titre externe
                un paragraph externe
```

main.cpp

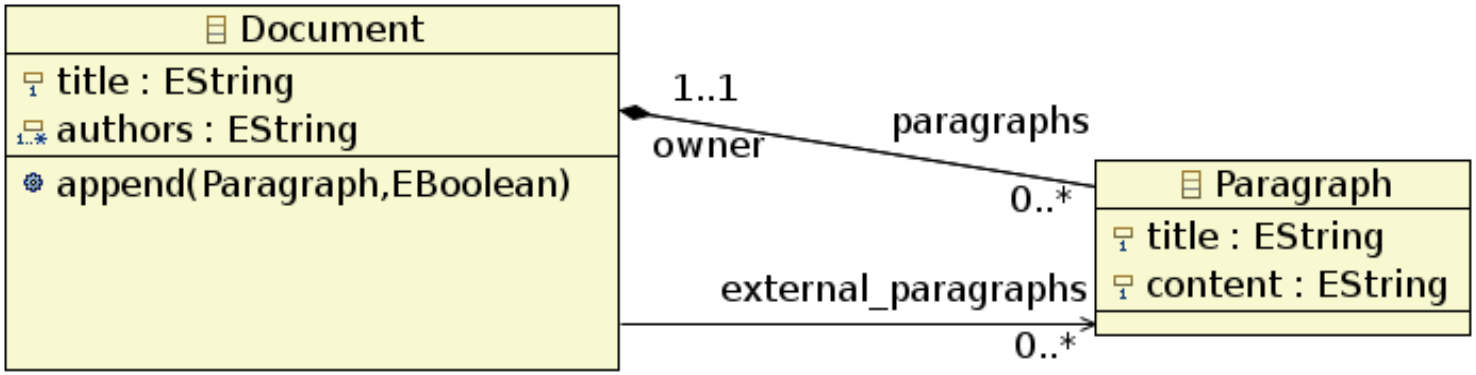
Document example: Destructor

→ Is there something to destroy ??



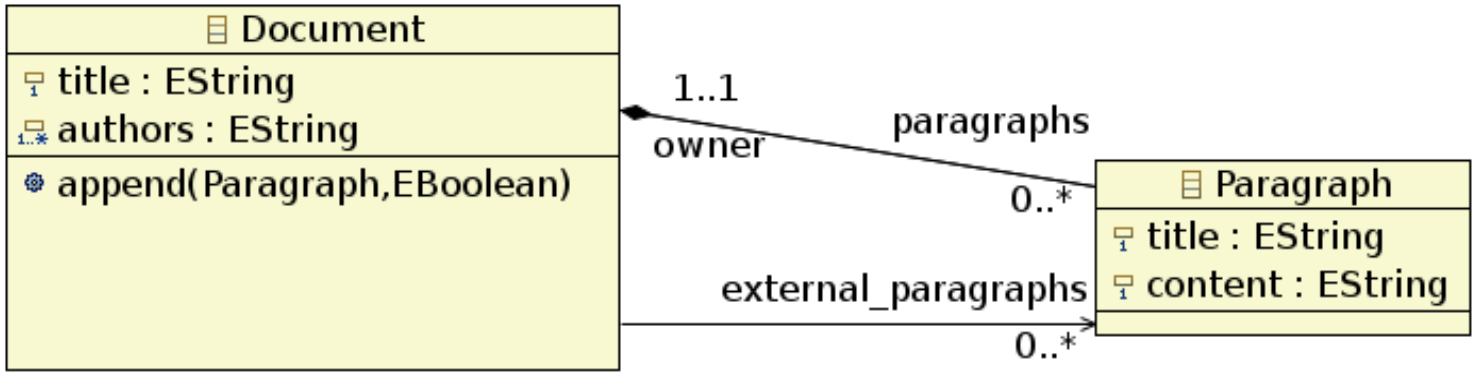
Document example: Destructor

→ Is there something to destroy ??



→ It depends on the implementation...

Document example: remember slide 8



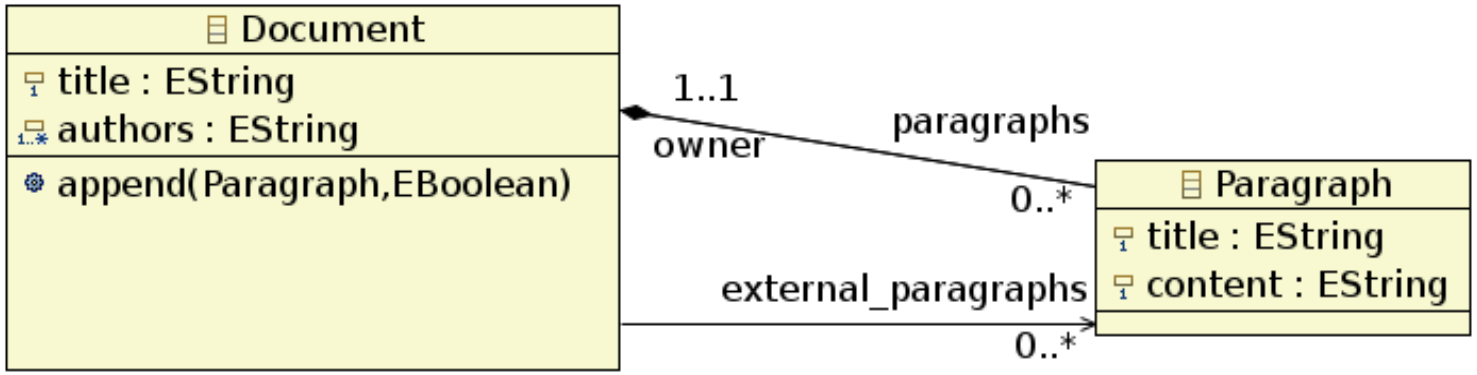

```
class Document{
private:
  string _title;
  vector<string> _authors;
  vector<Paragraph> _paragraphs;
  vector<Paragraph*> _external_paragraphs;
public:
  /*! constructors */
  Document(string title="default_t
         vector<string> new_authors
          vector<Paragraph> new_par
           vector<Paragraph*> new_e

  void append(Paragraph&, bool);
};
```

Note the difference between containment and association

There are various points of view on this point. You can choose your point of view but you will have to motivate your choice.

For me, when there is a containment, the object life is under the responsibility of the container. Otherwise it is not.

Document example: another implementation




```
class Document{
private:
  string _title;
  vector<string> _authors;
  vector<Paragraph*> _paragraphs;
  vector<Paragraph*> _external_paragraphs;
public:
  /*! constructors */
  Document(string title="default_title",
         vector<string> new_authors= vector<string>(),
          vector<Paragraph> new_paragraphs=vector<Paragraph>(),
           vector<Paragraph*> new_external_paragraphs=vector<Paragraph*>());

  void append(Paragraph&, bool);
};
```

Document example: another implementation

```
#include "document.h"


void Document::append(Paragraph& p, bool owned)
{
  if (owned == true)
  {
     _paragraphs.push_back(new Paragraph(p));
  }
  else
  {
     _external_paragraphs.push_back(&p);
  }
}
```

A new place is reserved in the memory and the corresponding pointer is put in the container

Document example: another implementation

A new place is reserved in the memory
and the corresponding pointer is put
in the container

The previous statement is always true:
For me, when there is a containment, the object life is under the responsibility of the container.
Otherwise it is not.

```
#include "document.h"


void Document::append(Paragraph& p, bool owned)
{
  if (owned == true)
  {
     _paragraphs.push_back(new Paragraph(p));
  }
  else
  {
     _external_paragraphs.push_back(&p);
  }
}
```

Document example: another implementation

```
void Document::append(Paragraph& p, bool owned)
{
  if (owned == true)
  {
     _paragraphs.push_back(new Paragraph(p));
  }
  else
  {
     _external_paragraphs.push_back(&p);
  }
}

Document::~Document()
{
   for (Paragraph* ptr_p: paragraphs)
   {
      delete ptr_p;
   }
}
```

A new place is reserved in the memory and the corresponding pointer is putted in the container

The previous statement is always true:
For me, when there is a containment, the object life is under the responsibility of the container.
Otherwise it is not.

Consequently, the release of the memory is handled in destructor

Document example: another implementation

```
void Document::append(Paragraph& p, bool owned)
{
  if (owned == true)
  {
     _paragraphs.push_back(new Paragraph(p));
  }
  else
  {
     _external_paragraphs.push_back(&p);
  }
}

Document::~Document()
{
   for (Paragraph* ptr_p: paragraphs)
   {
      delete ptr_p;
   }
```

A new place is reserved in the memory and the corresponding pointer is putted in the container

The previous statement is always true:
For me, when there is a containment, the object life is under the responsibility of the container.
Otherwise it is not.

Consequently, the release of the memory is handled in destructor

The whole implementation must be consistent
→ dynamic creation everywhere
(copy constructor, assignment, etc) !

Document example: another implementation

```
void Document::append(Paragraph& p, bool owned)
{
  if (owned == true)
  {
     _paragraphs.push_back(new Paragraph(p));
  }
  else
  {
     _external_paragraphs.push_back(&p);
  }
}

Document::~Document()
{
   for (Paragraph* ptr_p: paragraphs)
   {
      delete ptr_p;
   }
```

A new place is reserved in the memory and the corresponding pointer is putted in the container

The previous statement is always true:
For me, when there is a containment, the object life is under the responsibility of the container.
Otherwise it is not.

Consequently, the release of the memory is handled in destructor

The whole implementation must be consistent
→ dynamic creation everywhere
(copy constructor, assignment, etc) !

In the remainder, we consider that paragraphs is a vector<Paragraph>

Document example: operator overloading

```
#include <string>
#include <vector>
#include <iostream>

class Document;
class Paragraph{
private:
  string _title;
  string _content;
  Document* _owner = nullptr;
public:
        //...
  //! assignment operator
  Paragraph& operator=(const Paragraph&);
  //! unary concatenation operator
  Paragraph& operator+=(const Paragraph&);

  //! binary concatenation operator
  friend Paragraph operator+(const Paragraph& p1, const Paragraph& p2);

  // and all the appropriate one....

  friend ostream& operator<<(ostream&, const Paragraph&);

};
```

paragraph.h

Document example: operator overloading

```
class Document{
private:
  string _title;
  vector<string> _authors;
  vector<Paragraph> _paragraphs;
  vector<Paragraph*> _external_paragraphs;
```

```
//...

Paragraph& Paragraph::operator=(const Paragraph& p)
{
  if (this == &p)
        return *this;
  this->_title = p._title;
  this->_content = p._content;
  this->_owner = p._owner;
return *this;
}

Paragraph& Paragraph::operator+=(const Paragraph& p)
{
  this->_title += (" / " + p._title);
  this->_content += p._content;
return *this;
}


Paragraph operator+(const Paragraph& p1, const Paragraph& p2) {
 return Paragraph(p1._title+" / "+p2._title, p1._content + p2._content);
}
```

Looks like the copy constructor but no new object is created !

We return (a reference on) the hidden argument (*this)

paragraph.cpp

Document example: operator overloading

```
Paragraph& Paragraph::operator=(const Paragraph& p)
{
  if (this == &p)
        return *this;
  _title = string(p._title);
  _content = string(p._content);
  _owner = p._owner;
return *this;
}
/*!
 *  operator +=, concatenate 'this' with another Paragraph.
 *  \ref _title is merged with p._title and separate by a slash ('/')
 *  \ref _content is simply merged with p._content without separation character
 *  \param p the Paragraph to concatenate with this
 */
Paragraph& Paragraph::operator+=(const Paragraph& p)
{
  _title += (" / " + p._title);
  _content += p._content;
return *this;
}

Paragraph operator+(const Paragraph& p1, const Paragraph& p2) {
 return Paragraph(p1._title+" / "+p2._title, p1._content + p2._content);
}
```

A choice is done on what is the semantic of the concatenation

→ documentation is important

paragraph.cpp

Document example: operator overloading

```
Paragraph& Paragraph::operator=(const Paragraph& p)
{
  if (this == &p)
         return *this;
  _title = p._title;
  _content = p._content;
  _owner = p._owner;
return *this;
}
Paragraph& Paragraph::operator+=(const Paragraph& p)
{
  _title += (" / " + p._title);
  _content += p._content;
return *this;
}
/*!
 *  operator +, concatenate two instances of Paragraph to create a new one.
 *  \ref _title is merged with p._title and separate by a slash ('/')
 *  \ref _content is simply merged with p._content without separation character
 *  \param p1 the first Paragraph to merge
 *  \param p2 the first Paragraph to merge
 *  \return Paragraph the resulting Paragraph, i.e. the merged one
 */
Paragraph operator+(const Paragraph& p1, const Paragraph& p2) {
 return Paragraph(p1._title+" / "+p2._title, p1._content + p2._content);
}
```

paragraph.cpp

Document example: operator overloading

```
Paragraph& Paragraph::operator=(const Paragraph& p)
{
  if (this == &p) return *this;
  _title = string(p._title);
  _content = string(p._content);
  _owner = p._owner;
return *this;
}
Paragraph& Paragraph::operator+=(const Paragraph& p)
{
  _title += (" / " + p._title);
  _content += p._content;
return *this;
}
/*!
 *  operator +, concatenate two ins
 *  \ref _title is merged with p._t
 *  \ref _content is simply merged with p._content without separation character
 *  \param p1 the first Paragraph to merge
 *  \param p2 the first Paragraph to merge
 *  \return Paragraph the resulting Paragraph, i.e. the merged one
 */
Paragraph operator+(const Paragraph& p1, const Paragraph& p2) {
 return Paragraph(p1._title+" / "+p2._title, p1._content + p2._content);
}
```

Note that this is a friend function and not a member function

paragraph.cpp

Document example: operator overloading

```
class Document{
private:
  string _title;
  vector<string> _authors;
  vector<Paragraph> _paragraphs;
  vector<Paragraph*> _external_paragraphs;
public:
  //! exception class
  class No_such_paragraph{};

  //...

  friend ostream& operator<<(ostream&, const Document&);

  Paragraph& operator[](unsigned int n); //throw(No_such_paragraph);

};

#endif
```

document.h

Document example: operator overloading

```
class Document{
private:
  string _title;
  vector<string> _authors;
  vector<Paragraph> _paragraphs;
  vector<Paragraph*> _external_paragraphs;
public:
  //! exception class
  class No_such_paragraph{};

  //...

  friend ostream& operator<<(ostream&, const Document&);

  Paragraph& operator[](unsigned int n); //throw(No_such_paragraph);

};

#endif
```

We should reuse existing `out_of_range` exception

document.h

Document example: operator overloading

```
  //...


Paragraph& Document::operator[](unsigned int n) throw(No_such_paragraph)
{
  if ( n >= (_paragraphs.size() + _external_paragraphs.size()) )
      throw No_such_paragraph();

  if ( n < _paragraphs.size() )
  {
    return _paragraphs.at(n);
  }
  else
  {
    return *(_external_paragraphs.at( n - _paragraphs.size() ));
  }

}
```

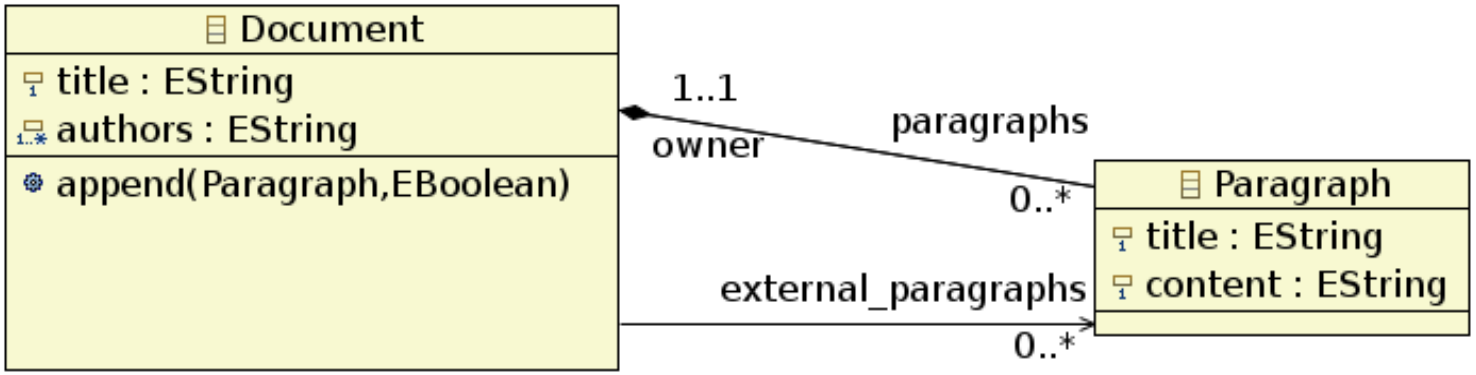
document.cpp

Document example: operator overloading

```
  //...


Paragraph& Document::operator[](unsigned int n) throw(No_such_paragraph)
{
  if ( n >= (_paragraphs.size() + _external_paragraphs.size()) )
      throw No_such_paragraph();

  if ( n < _paragraphs.size() )
  {
    return _paragraphs.at(n);
  }
  else
  {
    return *(_external_paragraphs.at( n - _paragraphs.size() ));
  }

}
```

document.cpp

Document example: critics and problems



- What if an external_paragraphs is delete by its owner ?
  - No easy way to know it
  - Can lead to a segmentation fault
  - Require the use of smart pointer (not part of this course)
    - http://en.cppreference.com/w/cpp/memory
- What if we want to add chapter, section, sub-section, ...
- What if we want to interleave internal and external chapters...