(Minimal)
(Practical)
Introduction to **C++** and **OOP**

For use in the Geant4 course

Largely incomplete

Not meant to replace good C++, UML, and OOP books!

Maria Grazia Pia

C++ basics

*C++ is **not** an object oriented language*

A “superset” of C

You can write procedural code in C++

Getting started

```
// my first program in C++
#include <iostream>
int main ()
{
  std::cout << "Hello World!";
  return 0;
}
```

// This is a **comment** line

#include <iostream>

- directive for the **preprocessor**

int main ()

- beginning of the definition of the **main function**
- the main function is the point by where all C++ programs start their execution
- all C++ programs must have a main function
- body enclosed in braces {}

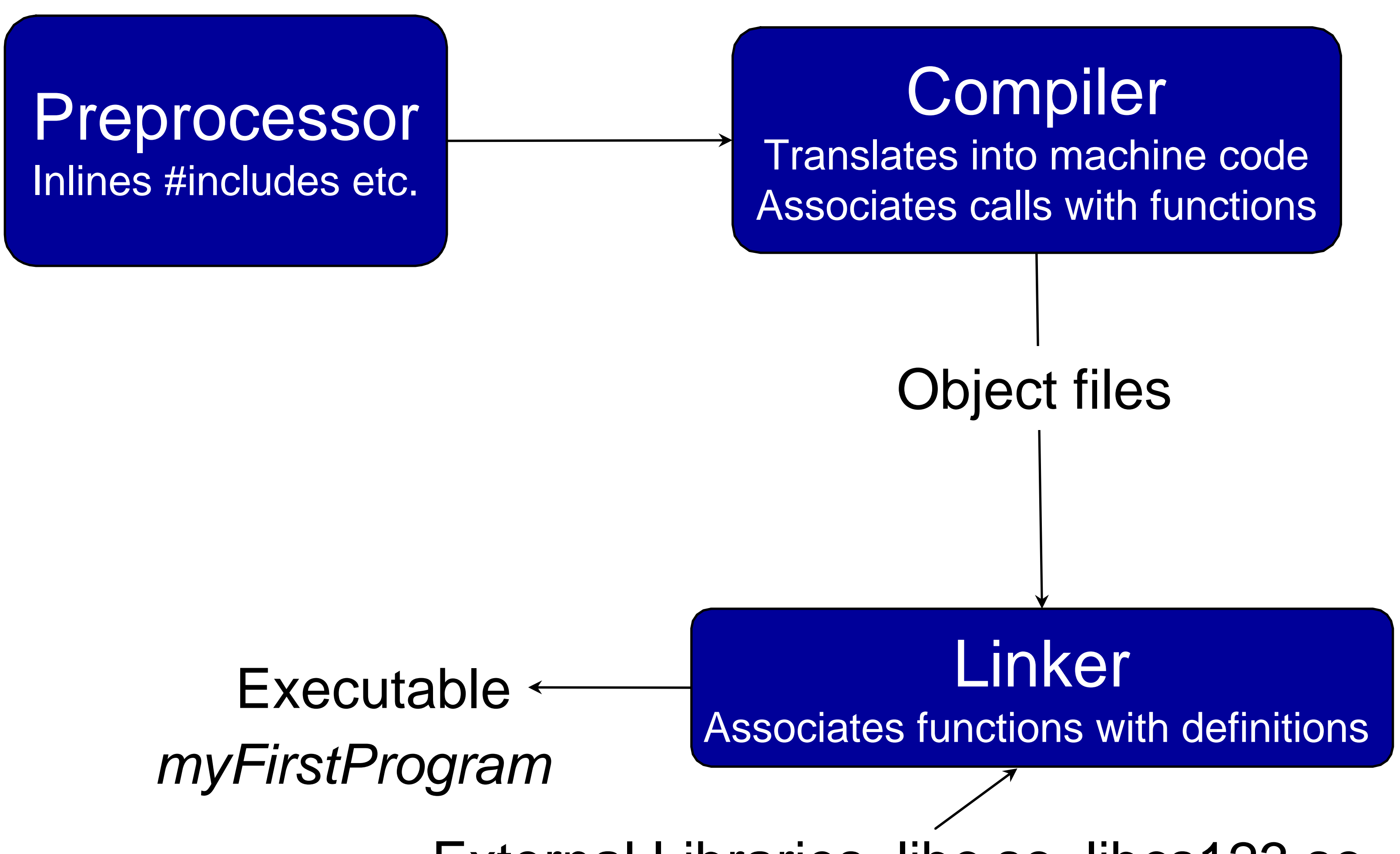
cout << "Hello World";

- C++ statement
- cout is declared in the iostream standard file within the std namespace
- cin
- semicolon (;) marks the end of the statement

return 0;

- the return statement causes the main function to finish
- return may be followed by a return code (here: 0)
  - return code 0 for the main function is generally interpreted as the program worked OK

Compilation



Using namespace

```
#include <iostream>
#include <string>
...
std::string question = “What do I learn this week?”;
std::cout << question << std::endl;

using namespace std;
…
string answer = “How to use Geant4”;
cout << answer << endl;
```

Variables

```
#include <iostream>
#include <string>
using namespace std;
int main ()
{
  // declaring variables:
  int a, b; // declaration
  int result;
  // process:
  a = 5;
  b = 2;
  a = a + 1;
  result = a - b;
  // print out the result:
  cout << result << endl;
  string myString = "This is a string";
  cout << myString << endl;
  const int neverChangeMe = 100;
  // terminate the program:
  return 0;
}
```

Maria Grazia Pia

Scope of variables

- global variables can be referred from anywhere in the code
- local variables: limited to the block enclosed in braces ({})

Initialization

```
int a = 0; // assignment operator
int a(0);  // constructor
```

const

the value cannot be modified after definition

References and Pointers

The address that locates a variable within memory is what we call a ***reference*** to that variable

```
x = &y;        // reference operator & "the address of"
```

Reference is an alias

```
int i = 10;
int& ir = i;        // reference (alias)
ir = ir + 1;        // increment i
```

A variable which stores a reference to another variable is called a **pointer**
Pointers are said to "point to" the variable whose reference they store

```
z = *x;        // z equal to "value pointed by" x
```

```
double* z;   // z is a pointer to a double
double x = 35.7;
z = &x;      // therefore *z is 35.7
```

```
z = 0;       // null pointer (not pointing to any valid reference or memory address)
```

Read pointer declarations right to left

```
// A const River
const River nile;

// A pointer to a const River
const River* nilePc;

// A const pointer to a River
River* const nileCp;

// A const pointer to a const River
const River* const nileCpc;
```

Dynamic memory

Operator new

pointer = new type

```
Student* paul = new Student;
```

If the allocation of this block of memory failed,
the failure could be detected by checking if paul took a null pointer value:

```
if (paul == 0) {
  // error assigning memory, take measures
};
```

Operator delete

```
delete paul;
```

Dynamic memory should be freed once it is no longer needed,
so that the memory becomes available again for other requests of dynamic memory
Rule of thumb: every **new** must be paired by a **delete**
Failure to free memory: **memory leak**

C++ Gotcha

Do not return pointers (or references) to local variables!

```
double* myFunction(void) {
  double d;
  return &d;
}
int main() {
  double* pd = myFunction();
  *pd = 3.14;
  return 0;
}
```

Boom! (maybe)

C++ “Gotcha”

Uninitialized pointers are bad!

```
int* i;

if ( someCondition ) {
    …
    i = new int;
} else if ( anotherCondition ) {
    …
    i = new int;
}

*i = someVariable;
```

“null pointer exception”

Memory allocation jargon

- **"on the stack"**
  - scope: block delimited by {}
  - object alive till it falls out of scope
  - calls constructor / destructor
- **"on the heap"**
  - **new** and **delete** operators
  - new calls constructor, delete calls destructor
  - object exists independently of scope in which it was created
  - also "on the free store" or "allocated in dynamic memory"
  - be careful: new → delete, new[] → delete[]
  - for safety, same object should both allocate and deallocate

Operators
(most common ones)

Assignment =

Arithmetic operators +, -, *, /, %

Compound assignment +=, -=, *=, /=, …

`a+=5; // a=a+5;`

Increase and decrease ++, --

`a++; // a=a+1;`

Relational and equality operators ==, !=, >, <, >=, <=

Logical operators ! (not), && (and), || (or)

Conditional operator (?)

```
a>b ? a : b
// returns whichever is greater, a or b
```

Explicit type casting operator

`int i; float f = 3.14; i = (int) f;`

Control structures

```
if (x == 100)
 {
   cout << "x is ";
   cout << x;
 }
```

```
if (x == 100)
  cout << "x is 100";
else
  cout << "x is not 100";
```

```
while (n>0) {
  cout << n << ", ";
  --n;
}
```

```
do {
  cout << "Enter number (0 to end): ";
  cin >> n;
  cout << "You entered: " << n << endl;
} while (n != 0);
```

for *(initialization; condition; increase) statement;*

```
for (int n=10; n>0; n--) {
  cout << n << ", ";
 }
```

```
for (n=10; n>0; n--)
 {
   cout << n << ", ";
   if (n==3)
    {
      cout << "countdown aborted!";
      break;
    }
 }
```

```
loop:
cout << n << ", ";
n--;
if (n>0) goto loop;
cout << “Procedural programming!“ ;
```

Functions

In C++ *all* function parameters are passed by **copy**

```
Type name(parameter1, parameter2, ...)
{
  statements…;
  return somethingOfType;
}
```

No type: void

```
void printMe(double x)
{
  std::cout << x << std::endl;
}
```

Arguments passed by value and by reference

```
int myFunction (int first, int second);
```

Pass a **copy** of parameters

```
int myFunction (int& first, int& second);
```

Pass a **reference** to parameters

They may be **modified** in the function!

```
int myFunction (const int& first, const int& second);
```

Pass a **const reference** to parameters

They may **not** be **modified** in the function!

More on Functions

Default values in parameters

```
double divide (double a, double b=2. )
{
  double r;
  r = a / b;
  return r;
}
```

```
int main ()
{
  cout << divide (12.) << endl;
  return 0;
}
```

Overloaded functions

Same name, different parameter type

A function cannot be overloaded only by its return type

```
int operate (int a, int b)
{
  return (a*b);
}
```

```
double operate (double a, double b)
{
  return (a/b);
}
```

OOP basics

OOP basic concepts

- **Object, Class**
  - A class defines the abstract characteristics of a thing (object), including the thing's attributes and the thing's behaviour
- **Inheritance**
  - "Subclasses" are more specialized versions of a class, which ***inherit*** attributes and behaviours from their parent classes (and can introduce their own)
- **Encapsulation**
  - Each object exposes to any class a certain ***interface*** (i.e. those members accessible to that class)
  - Members can be **public**, **protected** or **private**
- **Abstraction**
  - Simplifying complex reality by modelling classes appropriate to the problem
  - One works at the most appropriate level of inheritance for a given aspect of the problem
- **Polymorphism**
  - It allows one to treat derived class members just like their parent class' members

Class and Object

Object: is characterized by
attributes (which define its state) and operations

A **class** is the blueprint of objects of the same type

```
class Rectangle {
  public:
  Rectangle (double,double); // constructor
  ~Rectangle() { // empty; } // destructor
  double area () { return (width * height); } // member function
  private:
  double width, height; // data members
};
```

```
Rectangle  rectangleA (3.,4.); // instantiate an object of type "Rectangle"
Rectangle* rectangleB = new Rectangle(5.,6.);
cout << "A area: "  <<  rectangleA.area()   << endl;
cout << "B area: "  <<  rectangleB->area() << endl;
delete rectangleB;                    // invokes the destructor
```

The class interface in C++

Usually defined in a header (.h or .hh) file:

```
class Car {
  public:
  // Members can be accessed by any object

  protected:
  // Can only be accessed by Car and its derived objects

  private:
  // Can only be accessed by Car for its own use.
};
```

Constructor and assignment

```
class String {
public:
  String( const char* value ); // constructor
  String( const String& rhs ); // copy constructor
  ~String();
  String& operator=( const String& rhs); // assignment operator
private:
        char* data;
};
```

```
int main() {
        String s1 = “anton”;
        String s2( “luciano” );
        s2 = s1;
};
```

Classes: Basic Design Rules

- Hide all member variables
- Hide implementation functions and data
- Minimize the number of public member functions
- Avoid default constructors
- Use **const** whenever possible / needed

OK:

A invokes a function of a B object
A creates an object of type B
A has a data member of type B

Bad:

A uses data directly from B
(without using B's interface)

Even worse:

A directly manipulates data in B

Inheritance

- A key feature of C++
- Inheritance allows to create classes derived from other classes
- Public inheritance defines an **“is-a”** relationship
  - *In other words: what applies to a base class applies to its derived classes*

```
class Base {
  public:
    virtual ~Base() {}
    virtual void f() {…}
  protected:
    int a;
  private:
    int b; …
};
```

```
class Derived : public Base {
 public:
    virtual ~Derived() {}
    virtual void f() {…}
    …
};
```

Polymorphism

- Mechanism that allows a derived class to modify the behaviour of a member declared in a base class

```
Base* b = new Derived;
b->f();
delete b;
```

Which f() gets called?

Liskov Substitution Principle

- One way of expressing the notion of subtype (or “is-a”)

If Derived is a subtype of Base, then
Base can be replaced everywhere with Derived,
without impacting any of the desired properties of the program

- In other words, you can substitute Base with Derived, and nothing will “go wrong”

Inheritance and virtual functions

```
class Shape
{
 public:
    Shape();
    virtual void draw();
};
```

A virtual function defines the interface and provides an implementation; derived classes may provide alternative implementations

```
class Circle : public Shape
{
    public:
        Circle (double r);
        void draw();
    private:
        double radius;
};
```

```
class Rectangle : public Shape
{
    public:
        Rectangle(double h, double w);
     private:
        double height, width;
};
```

Abstract classes, Abstract interfaces

Abstract class,
cannot be instantiated

```
class Shape
{
    public:
        Shape();
        virtual area() = 0;
};
```

Abstract Interface
a class consisting of
pure virtual functions only

A pure virtual function
defines the interface
and delegates the implementation
to derived classes

```
class Circle : public Shape
{
    public:
        Circle (double r);
        double area();
    private:
        double radius;
};
```

```
class Rectangle : public Shape
{
    public:
        Rectangle(double h, double w);
        double area();
    private:
        double height, width;
};
```

Concrete class

Inheritance and Virtual Functions

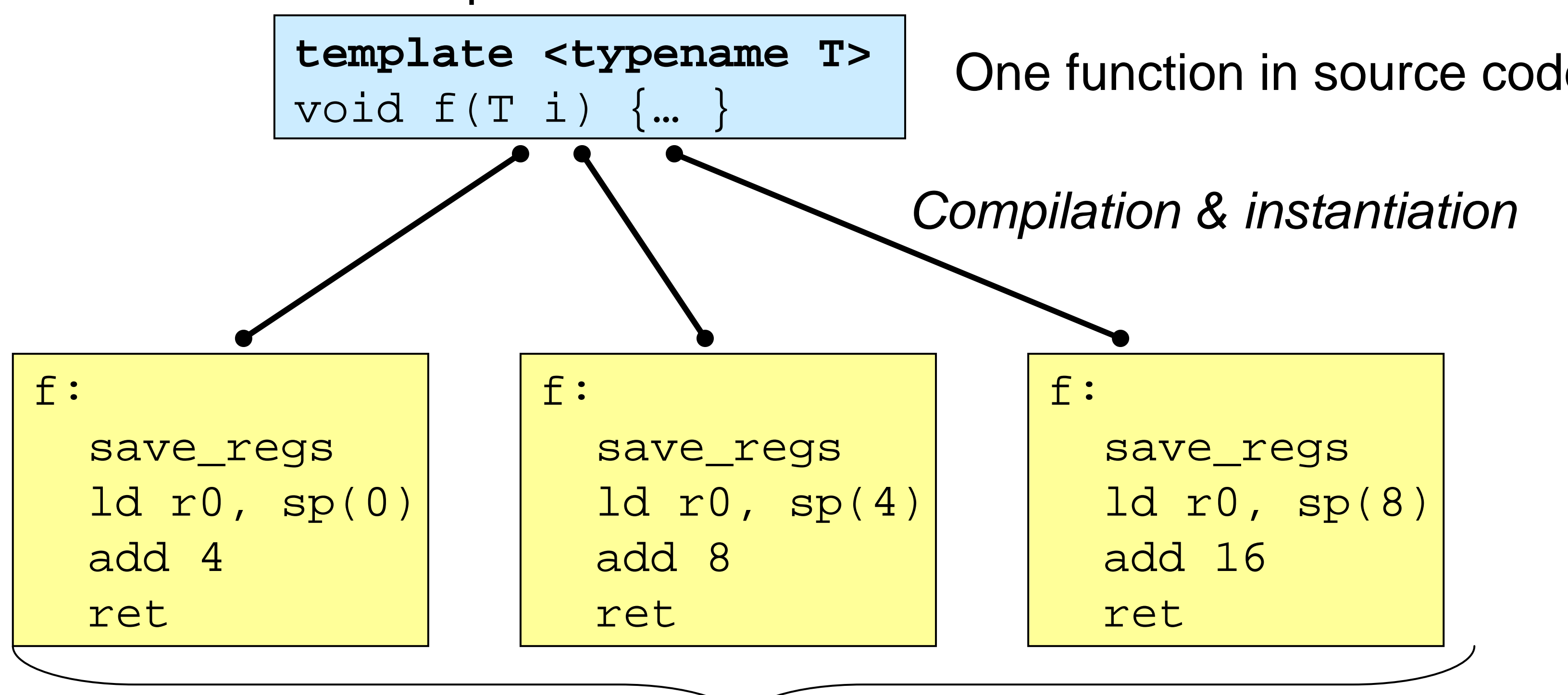
	Inheritance of the **interface**	Inheritance of the **implementation**
Non virtual function	Mandatory	Mandatory
Virtual function	Mandatory	By default Possible to reimplement
Pure virtual function	Mandatory	Implementation is mandatory

More C++

Templates

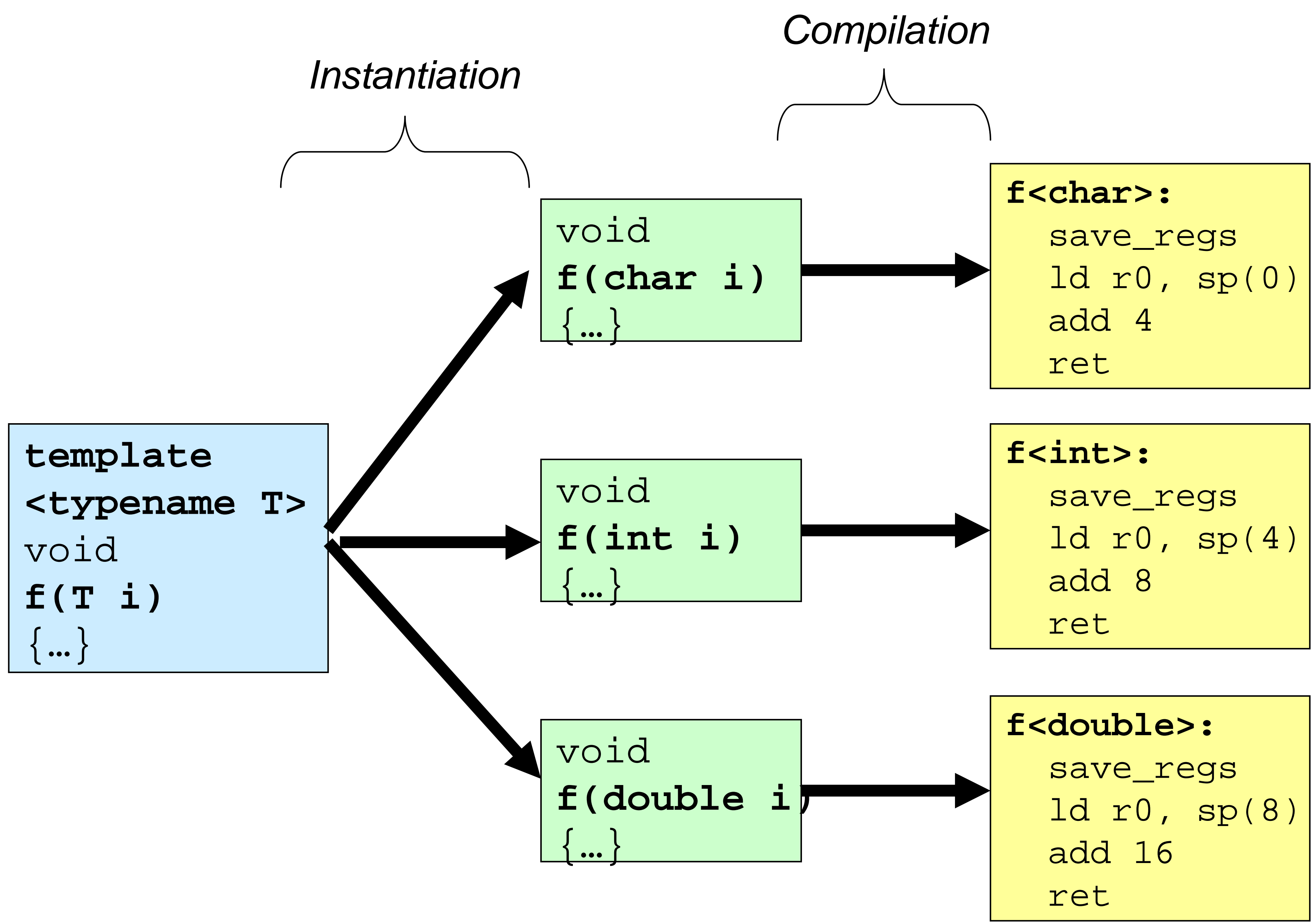
Minimal introduction, only to introduce STL

- A C++ template is just that, a template
- A single template serves as a pattern, so it can be used multiple times to create multiple instantiations



Multiple functions in assembly language

- **Function** templates
- **Class** templates
- **Member** templates


Instantiation
Compilation
template
<typename T>
void
f(T i)
{…}
void
f(char i)
{…}
void
f(int i)
{…}
void
f(double i)
{…}
f<char>:
save_regs
ld r0, sp(0)
add 4
ret
f<int>:
save_regs
ld r0, sp(4)
add 8
ret
f<double>:
save_regs
ld r0, sp(8)
add 16
ret

Standard Template Library (STL)

Containers

- **Sequence**
  - **vector**: array in contiguous memory
  - **list**: doubly-linked list (fast insert/delete)
  - **deque**: double-ended queue
  - stack, queue, priority queue
- **Associative**
  - **map**: collection of (key,value) pairs
  - **set**: map with values ignored
  - multimap, multiset (duplicate keys)
- **Other**
  - **string**, basic_string
  - **valarray:**for numeric computation
  - bitset: set of N bits

Algorithms

- **Non-modifying**
  - find, search, mismatch, count, for_each
- **Modifying**
  - copy, transform/apply, replace, remove
- **Others**
  - unique, reverse, random_shuffle
  - sort, merge, partition
  - set_union, set_intersection, set_difference
  - min, max, min_element, max_element
  - next_permutation, prev_permutation

std::string

Example:

```
#include <string>

void FunctionExample()
{
  std::string s, t;
  char c = 'a';
  s.push_back(c);  // s is now “a”;
  const char* cc = s.c_str();  // get ptr to “a”
  const char dd[] = ‘like’;
  t = dd;  // t is now “like”;
  t = s + t;  // append “like” to “a”
}
```

std::vector

use std::vector,
not built-in C-style array,
whenever possible

Example:

```
#include <vector>
void FunctionExample()
{
  std::vector<int> v(10);
  int a0 = v[3];         // unchecked access
  int a1 = v.at(3);      // checked access
  v.push_back(2);        // append element to end
  v.pop_back();          // remove last element
  size_t howbig = v.size();  // get # of elements
  v.insert(v.begin()+5, 2);  // insert 2 after 5th element
}
```

std::vector (more)

Example:

```
#include <vector>
#include <algorithm>

void FunctionExample()
{
  std::vector<int> v(10);
  v[5] = 3;  // set fifth element to 3
  std::vector<int>::const_iterator it
                = std::find(v.begin(), v.end(), 3);
  bool found = it != v.end();
  if (found) {
    int three = *it;
    }
}
```

Iterators

- *iterator* – kind of generalized pointer
- Each container has its own type of iterator

```
void FunctionExample() {
  stl::vector<int> v;
  stl::vector<int>::const_iterator it = v.begin();
  for (it = v.begin() ; it != v.end() ; it++) {
    int val = *it;
  }
}
```

A few practical issues

Organizational Strategy

image.hh

Header file: Class definition

```
void SetAllPixels(const Vec3& color);
```

image.cc

.cc file: Full implementation

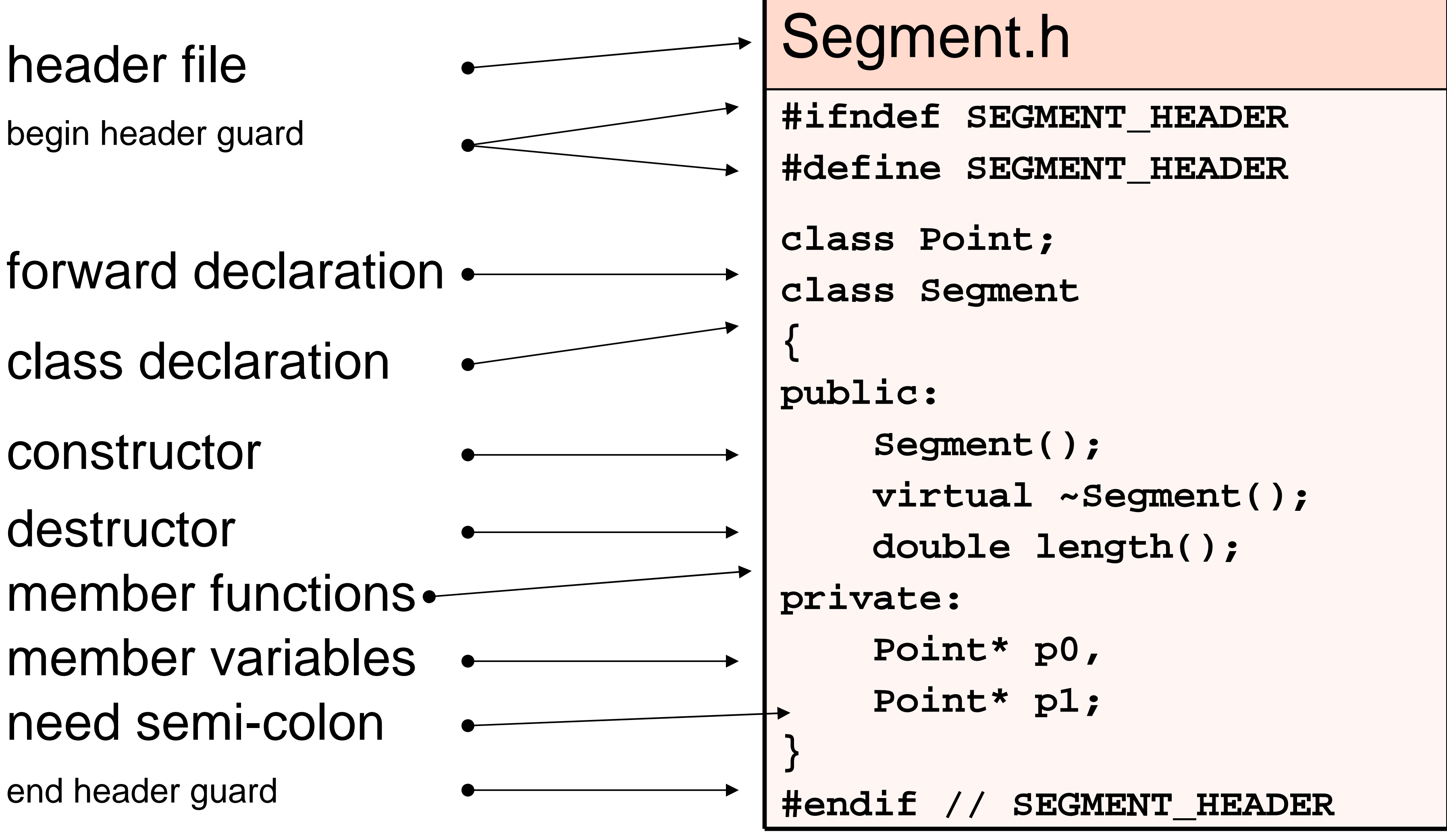
```
void Image::SetAllPixels(const Vec3& color) {
   for (int i = 0; i < width*height; i++)
     data[i] = color;
}
```

main.cc

Main function

```
myImage.SetAllPixels(clearColor);
```

How a Header File looks like



Forward Declaration

Gui.hh

```
Class Gui
{
//
};
```

Controller.hh

```
//Forward declaration
class Gui;

class Controller
{
//...
private:
    Gui* myGui;
//...
};
```

- In header files, only include what you must
- If only pointers to a class are used, use forward declarations

Header file and implementation

File Segment.hh

```
#ifndef SEGMENT_HEADER
#define SEGMENT_HEADER

class Point;
class Segment
{
public:
    Segment();
    virtual ~Segment();
    double length();
private:
    Point* p0,
    Point* p1;
};
#endif // SEGMENT_HEADER
```

File Segment.cc

```
#include “Segment.hh”
#include “Point.hh”
Segment::Segment() // constructor
{
    p0 = new Point(0.,0.);
    p1 = new Point(1.,1.);
}
Segment::~Segment() // destructor
{
    delete p0;
    delete p1;
}
double Segment::length()
{
    function implementation …
}
```

"Segmentation fault (core dumped)"

Typical causes:

```
int intArray[10];
intArray[10] = 6837;
```

Access outside of array bounds

```
Image* image;
image->SetAllPixels(colour);
```

Attempt to access a NULL or previously deleted pointer

These errors are often very difficult to catch and can cause erratic, unpredictable behaviour

UML

Unified Modelling Language

- The UML is a graphical language for
  - specifying
  - visualizing
  - constructing
  - documenting

  the artifacts of software systems
- Define an easy-to-learn, but semantically rich visual modeling language
- Added to the list of OMG adopted technologies in November 1997 as UML 1.1
- Version evolution

Building Blocks

- The basic building blocks of UML are:
  - model elements
    - classes, interfaces, components, use cases etc.
  - relationships
    - associations, generalization, dependencies etc.
  - diagrams
    - class diagrams, use case diagrams, interaction diagrams etc.
- Simple building blocks are used to create large, complex structures

Structural Modeling: Core Elements

Construct	Description	Syntax
class	a description of a set of objects that share the same attributes, operations, methods, relationships and semantics	
interface	a named set of operations that characterize the behavior of an element	«interface»
component	a physical, replaceable part of a system that packages implementation and provides the realization of a set of interfaces.	
node	a run-time physical object that represents a computational resource	
constraint	a semantic condition or restriction	{constraint}

Ma

Structural Modeling: Core Relationships

Construct	Description	Syntax
association	a relationship between two or more classifiers that involves connections among their instances	
aggregation	A special form of association that specifies a whole-part relationship between the aggregate (whole) and the component part	
generalization	a taxonomic relationship between a more general and a more specific element	
dependency	a relationship between two modeling elements, in which a change to one modeling element (the independent element) will affect the other modeling element (the dependent element)	
realization	a relationship between a specification and its implementation	

Class

- Classes can have 4 parts
  - **Name**
  - **Attributes**
  - **Operations**
  - Responsibilities

- Classes can show **visibility** and **types**

- All parts but the Name are optional

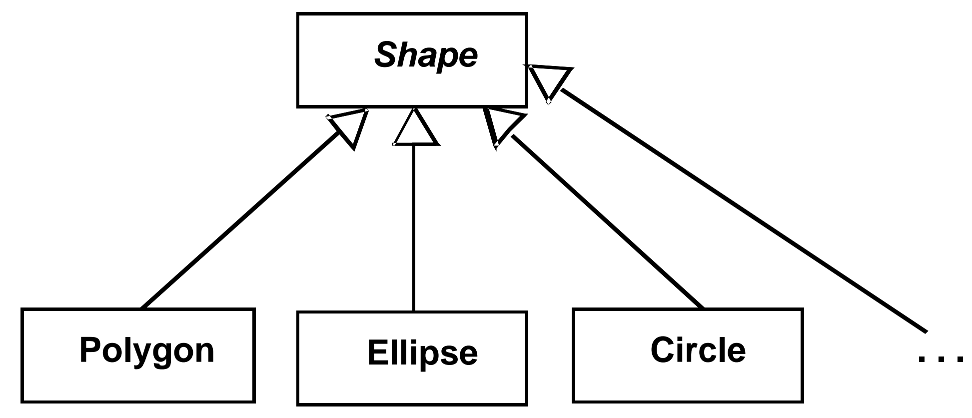
MyClassName

+Some**Public**Attribute : SomeType
-Some**Private**Attribute : SomeType
#Some**Protected**Attribute : SomeType

+ClassMethodOne()
+ClassMethodTwo()

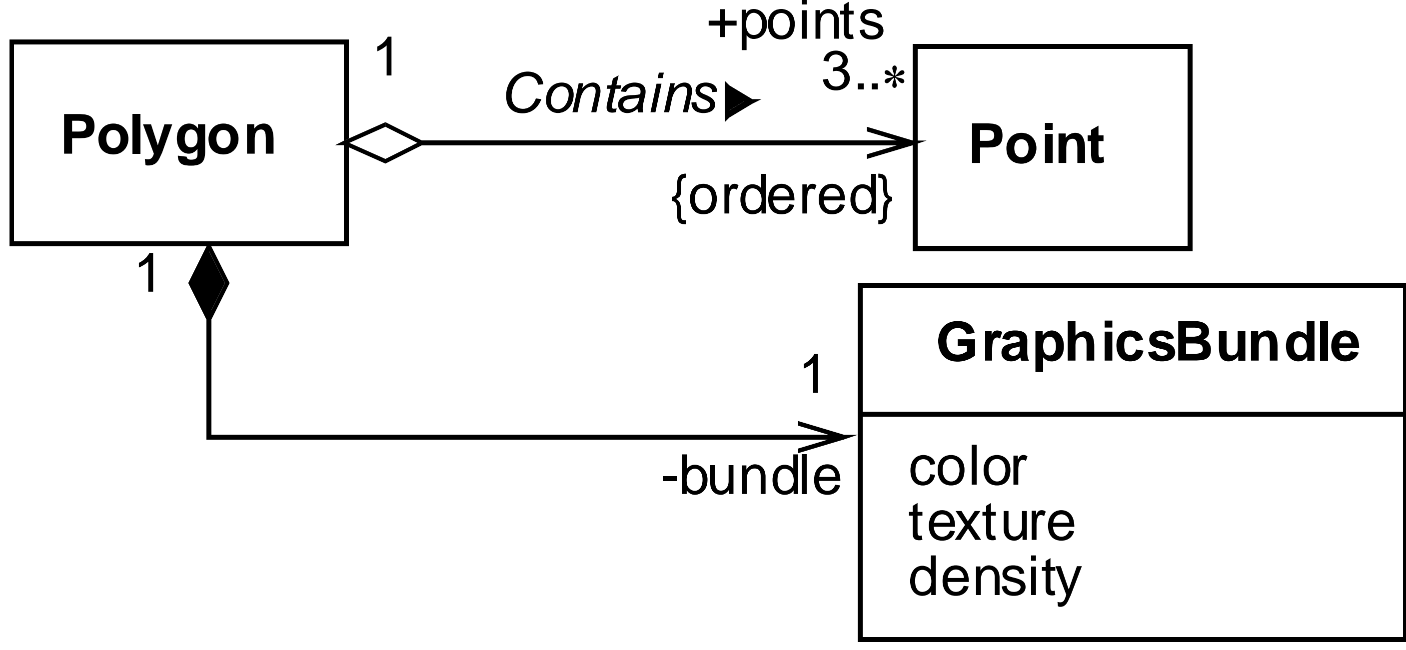
Responsibilities
-- can optionally be described here.

Generalization

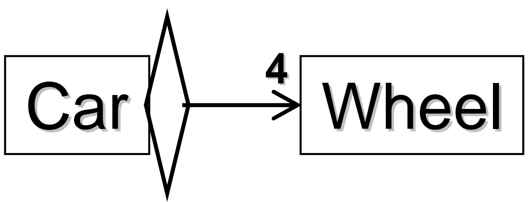


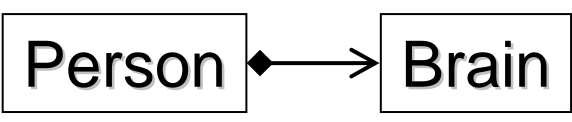
Models inheritance

Associations



Aggregation or Composition?





Aggregation is a relationship in which one object is a part of another

A aggregates B
=
B is part of A, but their lifetimes may be different

Composition is a relationship in which one object is an integral part of another

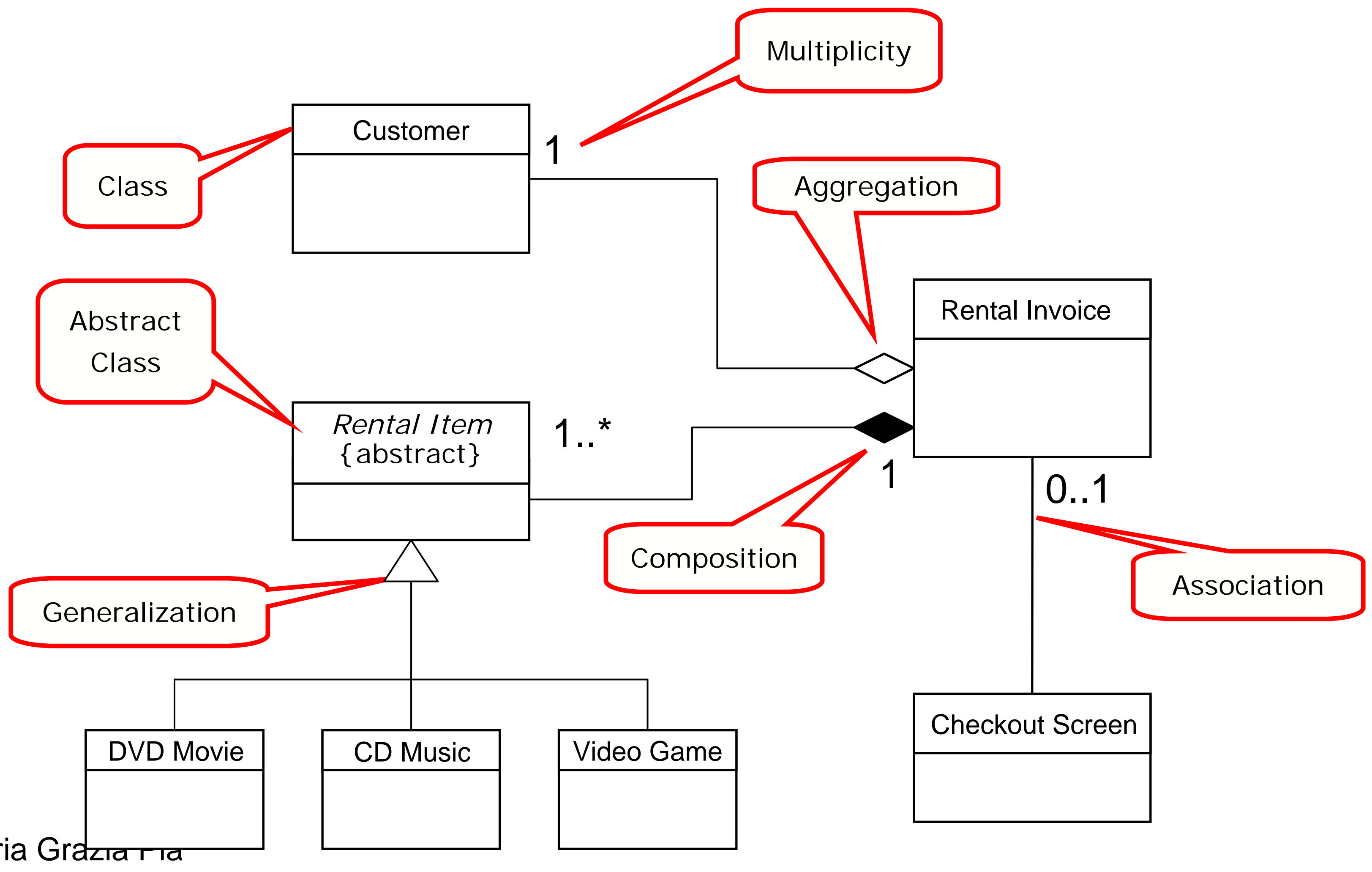
A contains B
=
B is part of A, and their lifetimes are the same

Main UML Diagrams

- **Class** Diagrams
- Use Case Diagrams
- **Collaboration** Diagrams
- **Sequence** Diagrams
- Package Diagrams
- Component Diagrams
- Deployment Diagrams
- Activity Diagrams
- State Diagrams

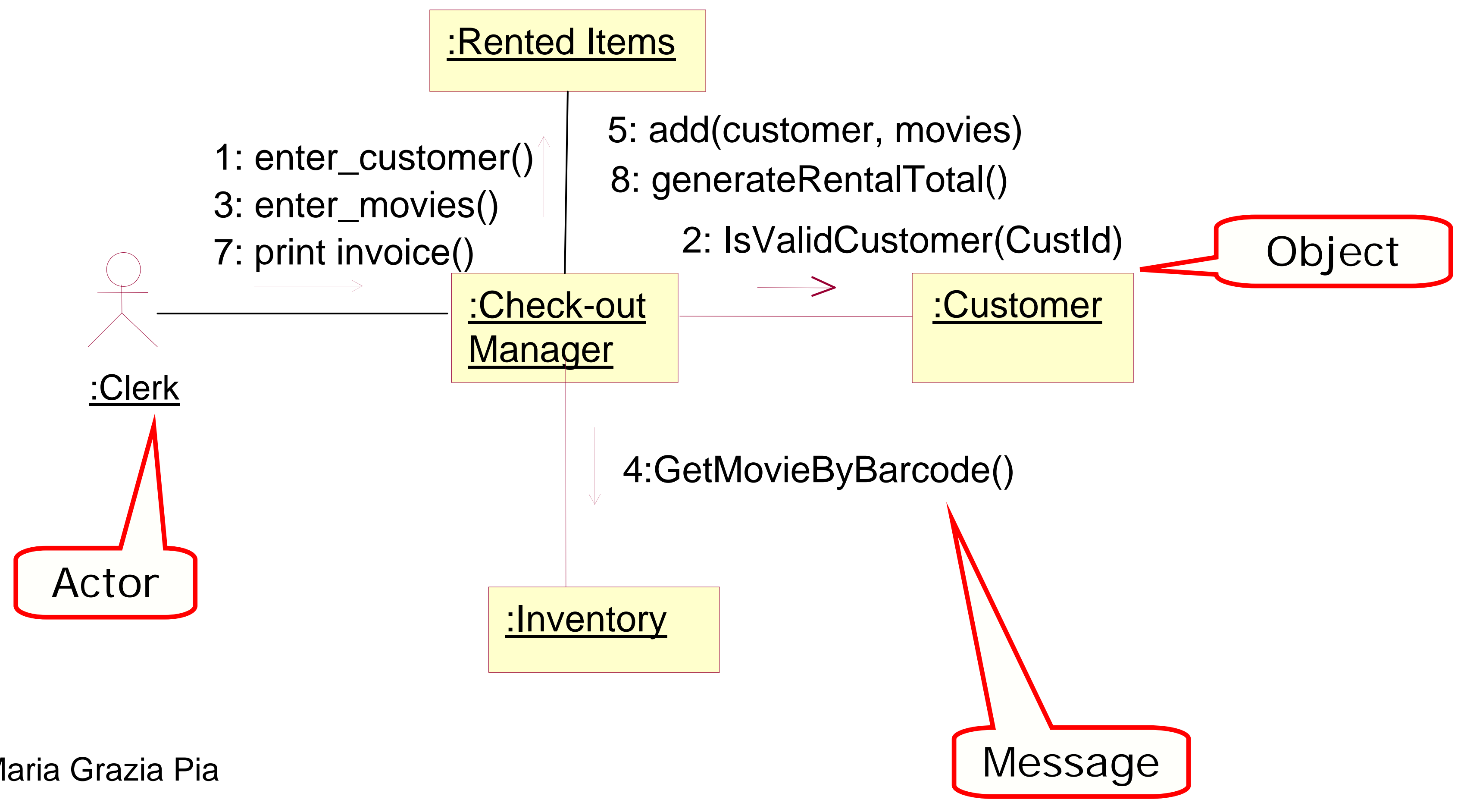
UML Class Diagram

Describe the **classes** in the system
and the **static** relationships between classes



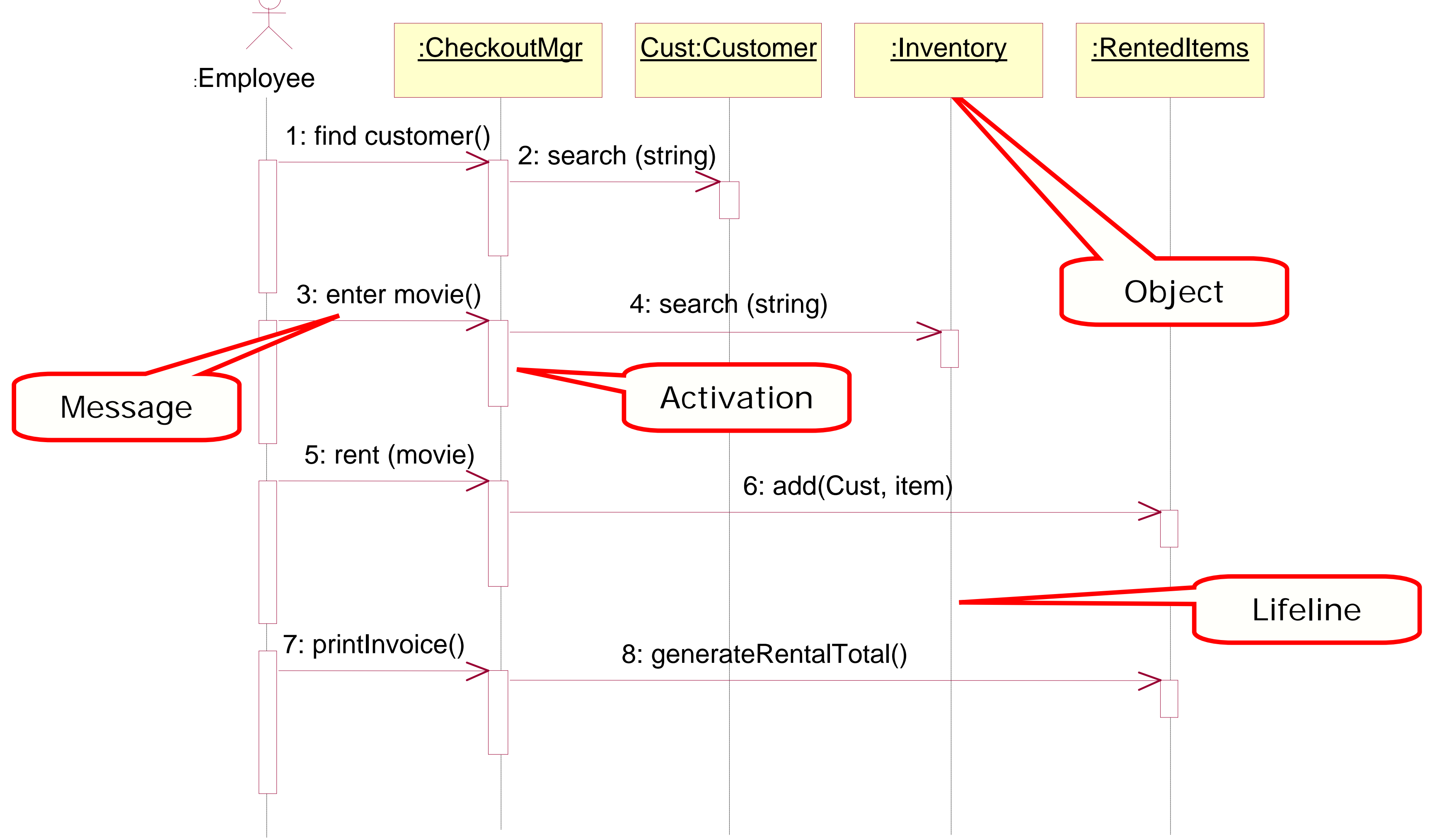
Collaboration Diagram - Rent Movie

Describe object **interactions** organized around the objects and their links to each other



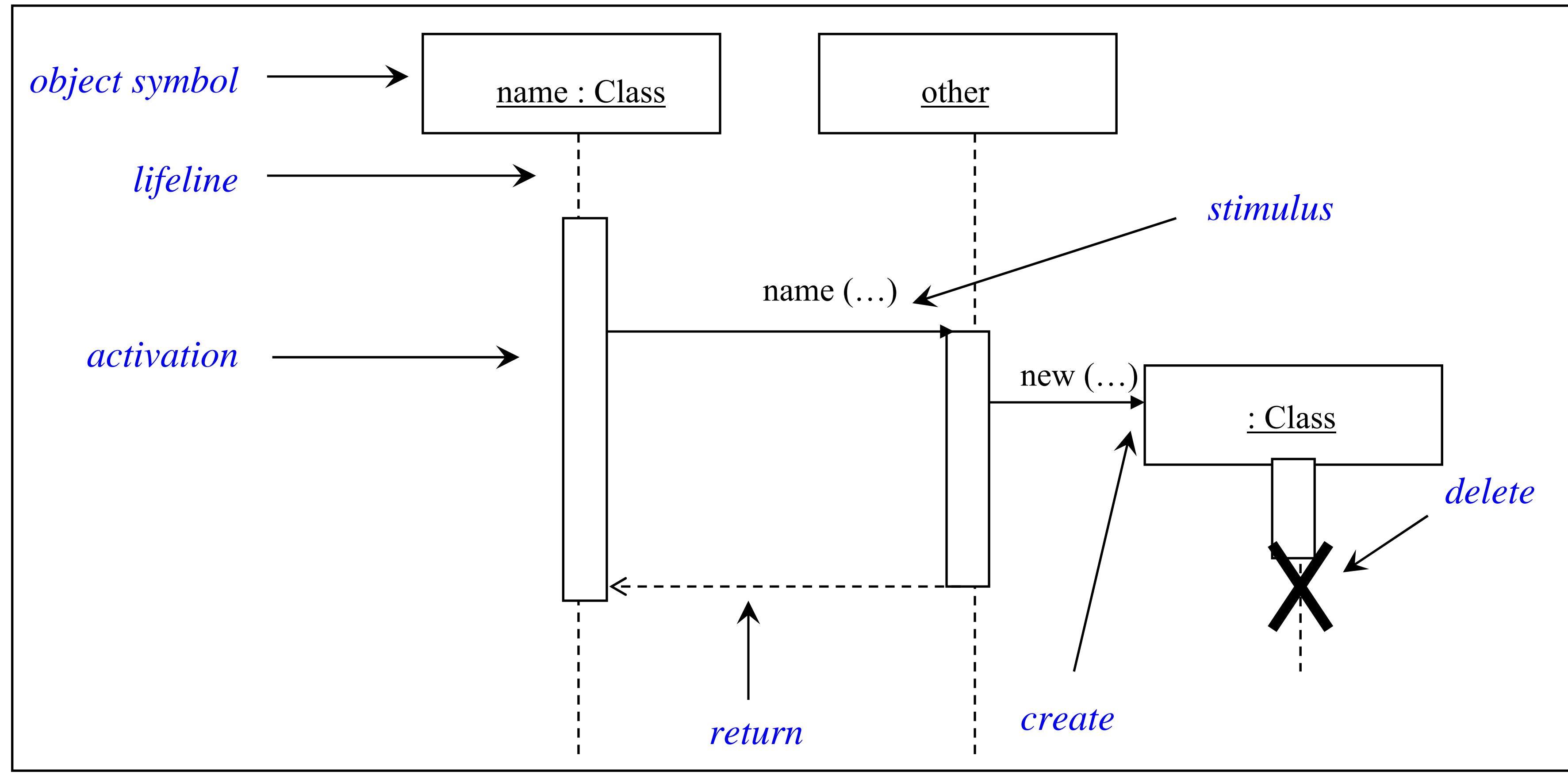
Sequence Diagram - Rent Movie

Describe interactions between objects arranged in time sequence



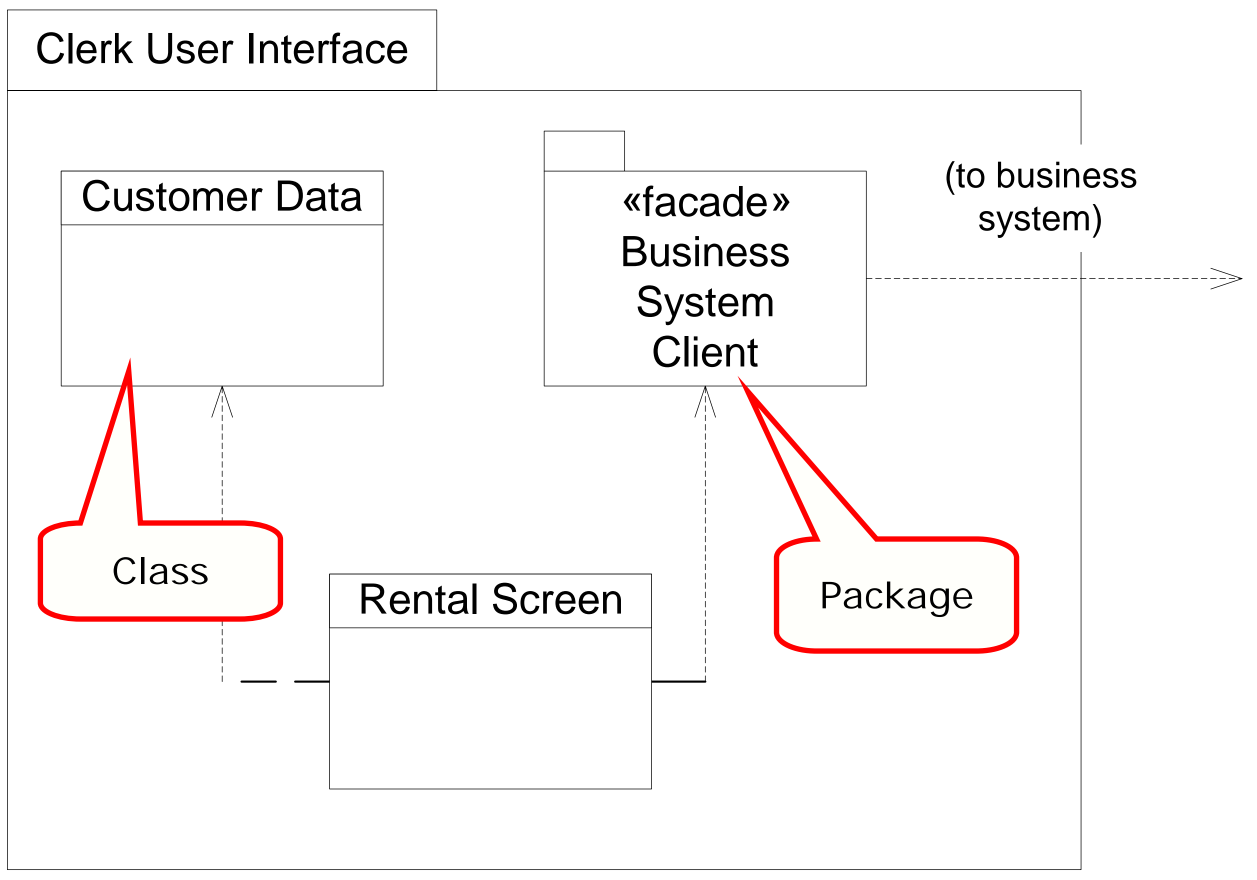
Sequence and collaboration diagrams can be cloned from each other

Sequence Diagram

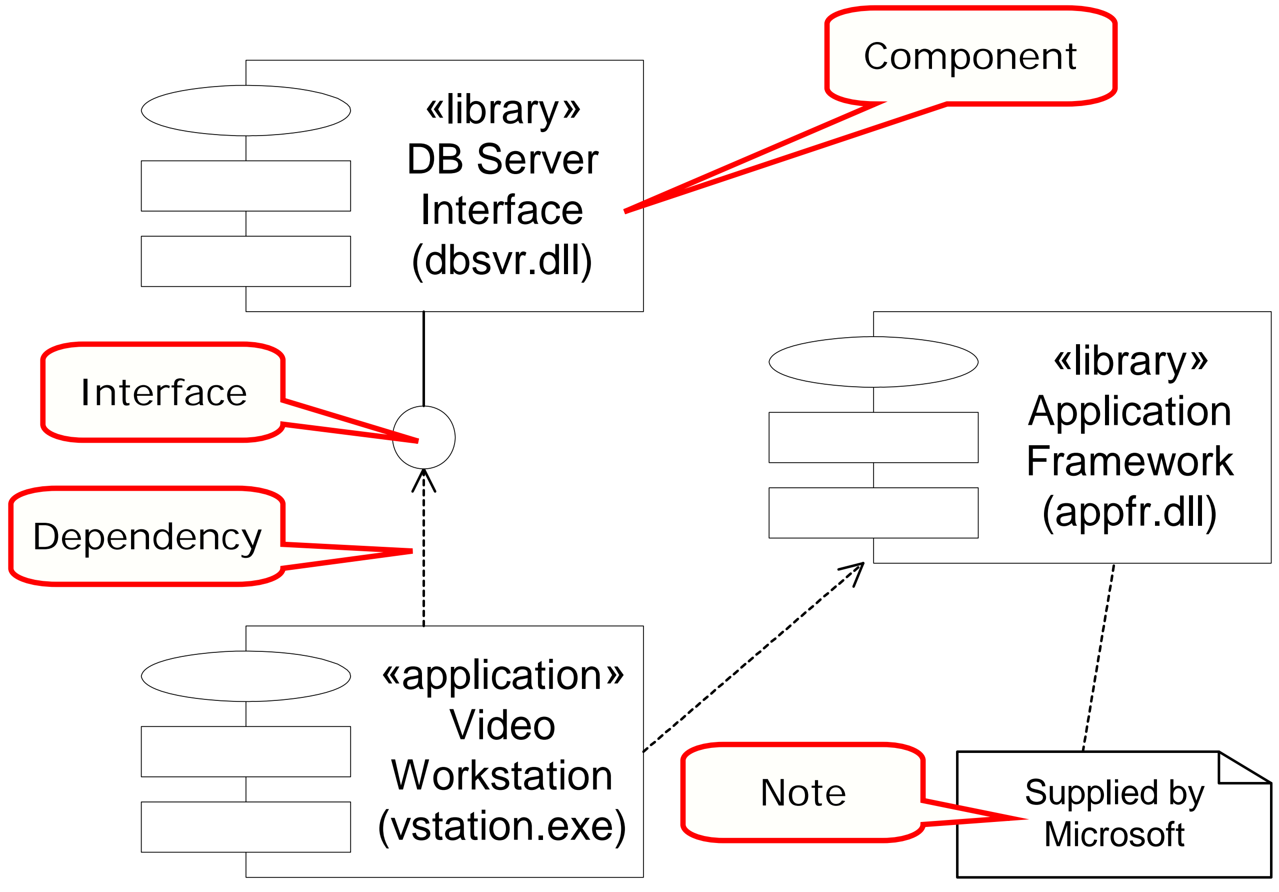


Maria Grazia Pia

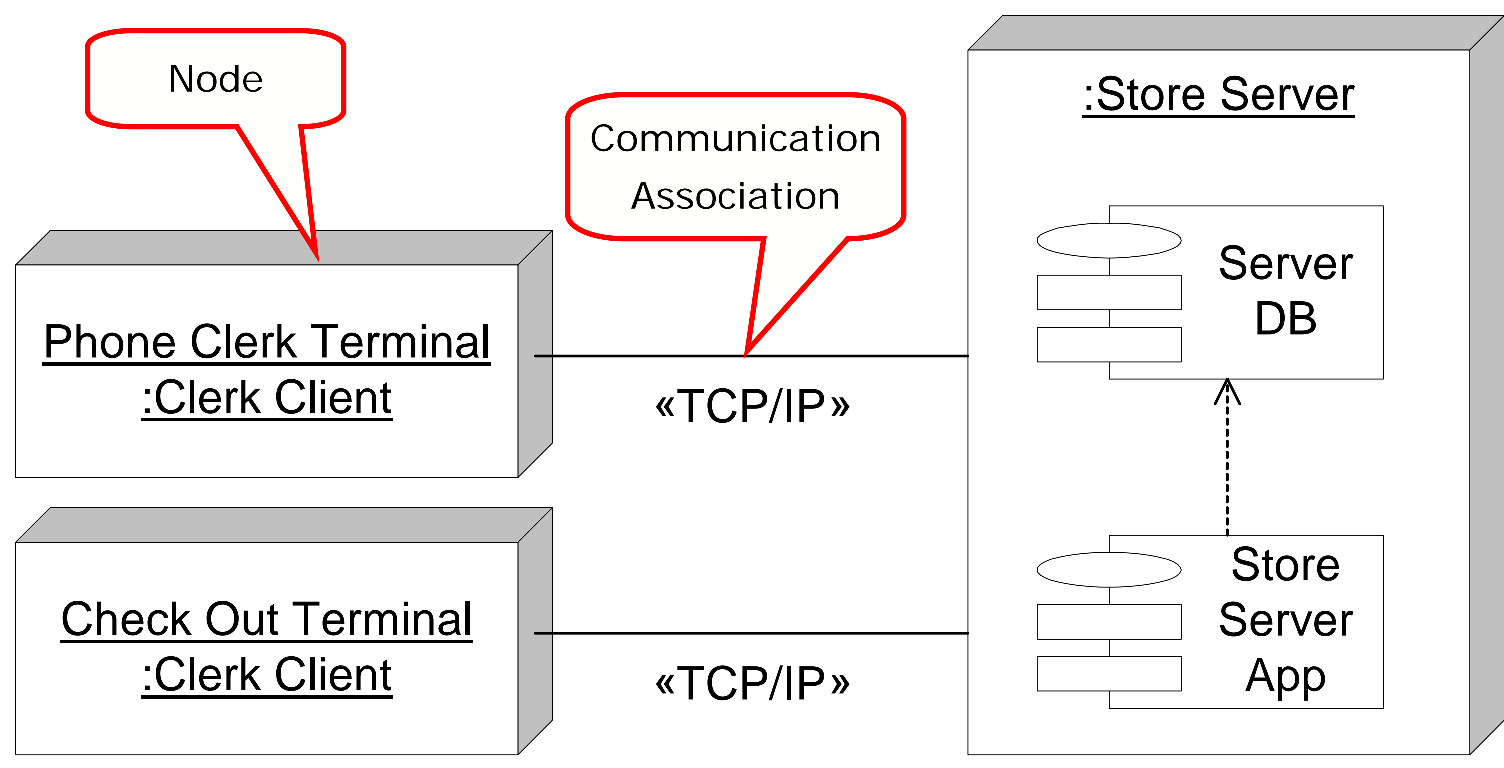
Package Diagram



Component Diagram



Deployment Diagram

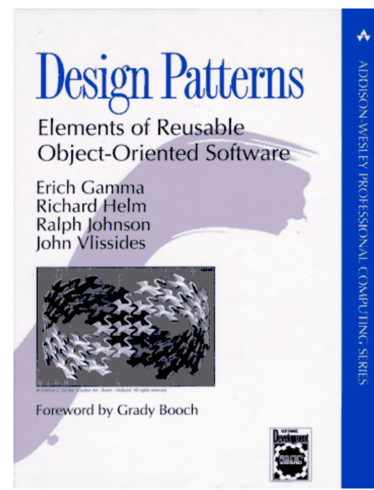


Introduction to Design Patterns

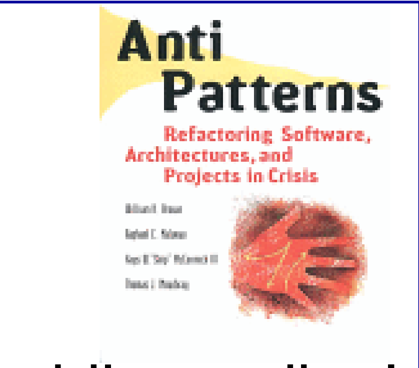
Maria Grazia Pia

Design Patterns

Gamma, Helm, Johnson and Vlissides
Design Patterns,
Addison-Wesley 1995, ISBN 0-201-63361-2
(Gang-of-Four)



- Each design pattern names, explains and evaluates an important and recurring design in object-oriented systems
- A design pattern makes it easier to reuse successful designs and architectures
- Three categories of patterns
  - Creational
  - Structural
  - Behavioral



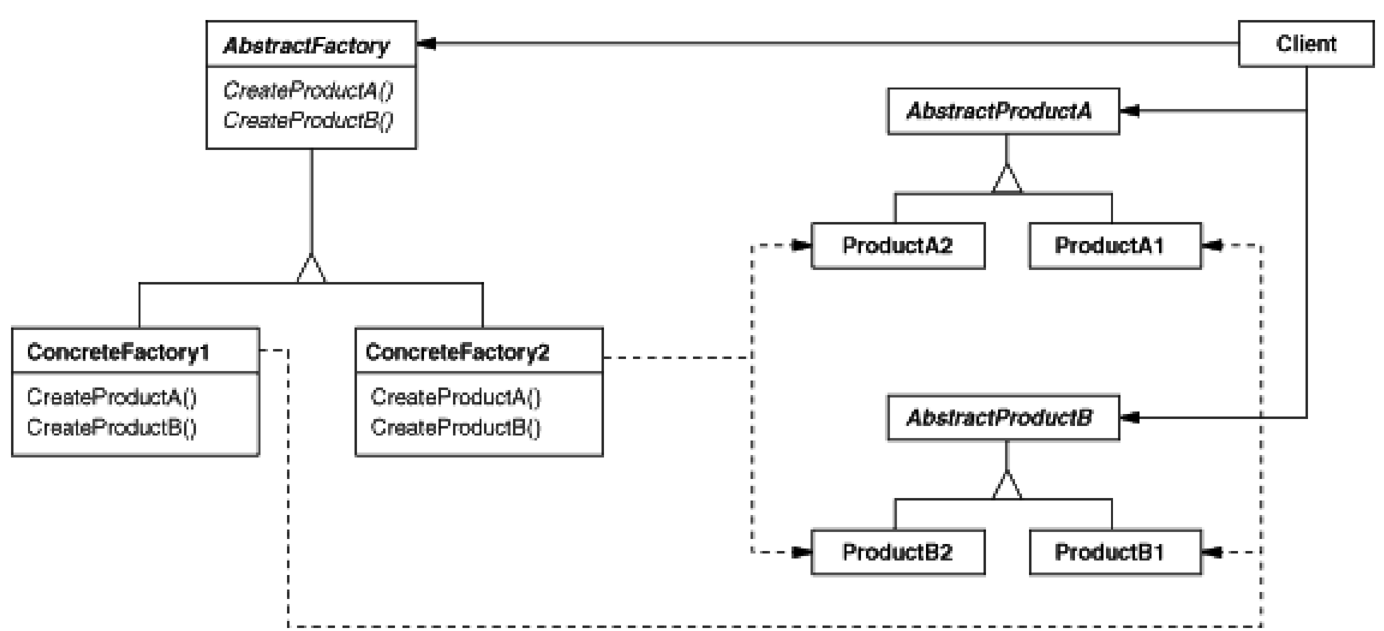
…also worthwhile reading!

List of design patterns

- Abstract Factory
- Adapter
- Bridge
- Builder
- Chain of Responsibility
- Command
- Composite
- Decorator
- Facade
- Factory Method
- Flyweight
- Interpreter
- Iterator
- Mediator
- Memento
- Observer
- Prototype
- Proxy
- Singleton
- State
- Strategy
- Template Method
- Visitor

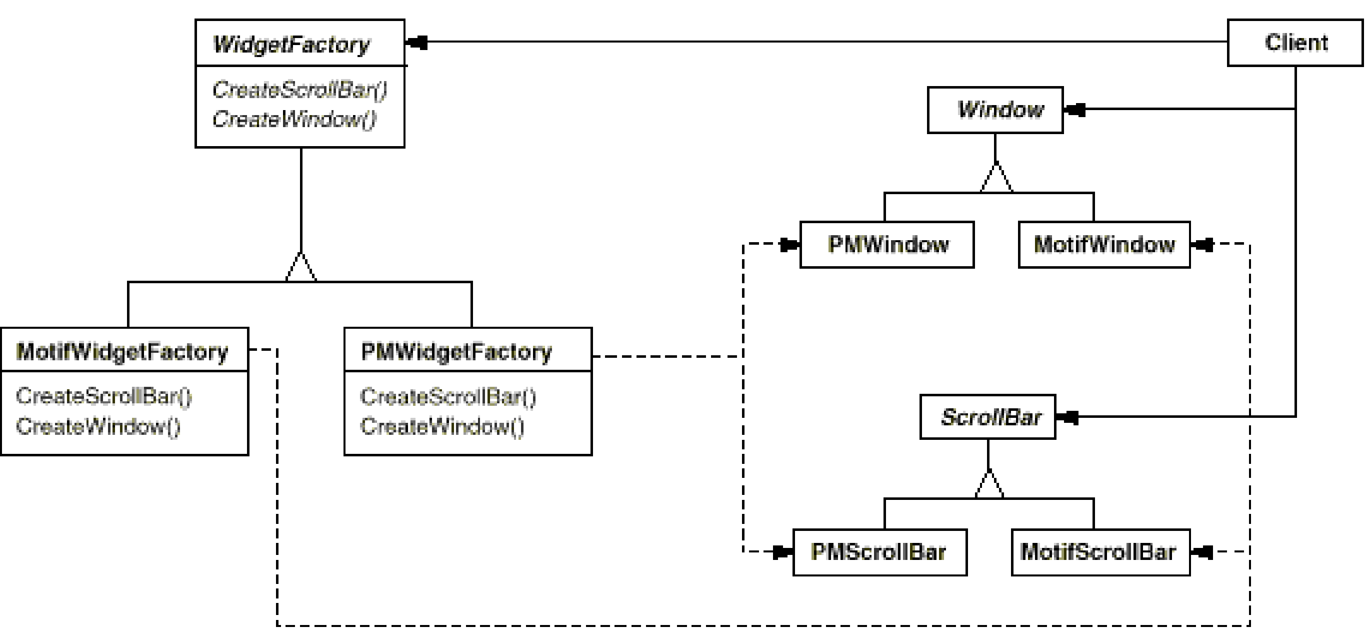
Abstract Factory

Provide an interface for creating families of related or dependent objects without specifying their concrete classes



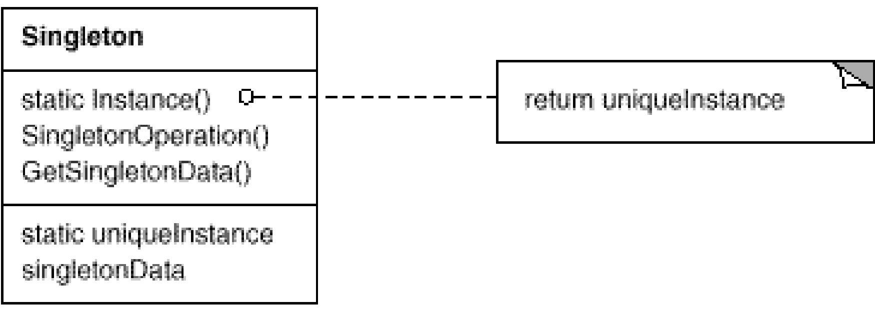
- AbstractFactory
  - declares an interface for operations that create product objects
- ConcreteFactory
  - implements the operations to create concrete product objects

Abstract Factory example

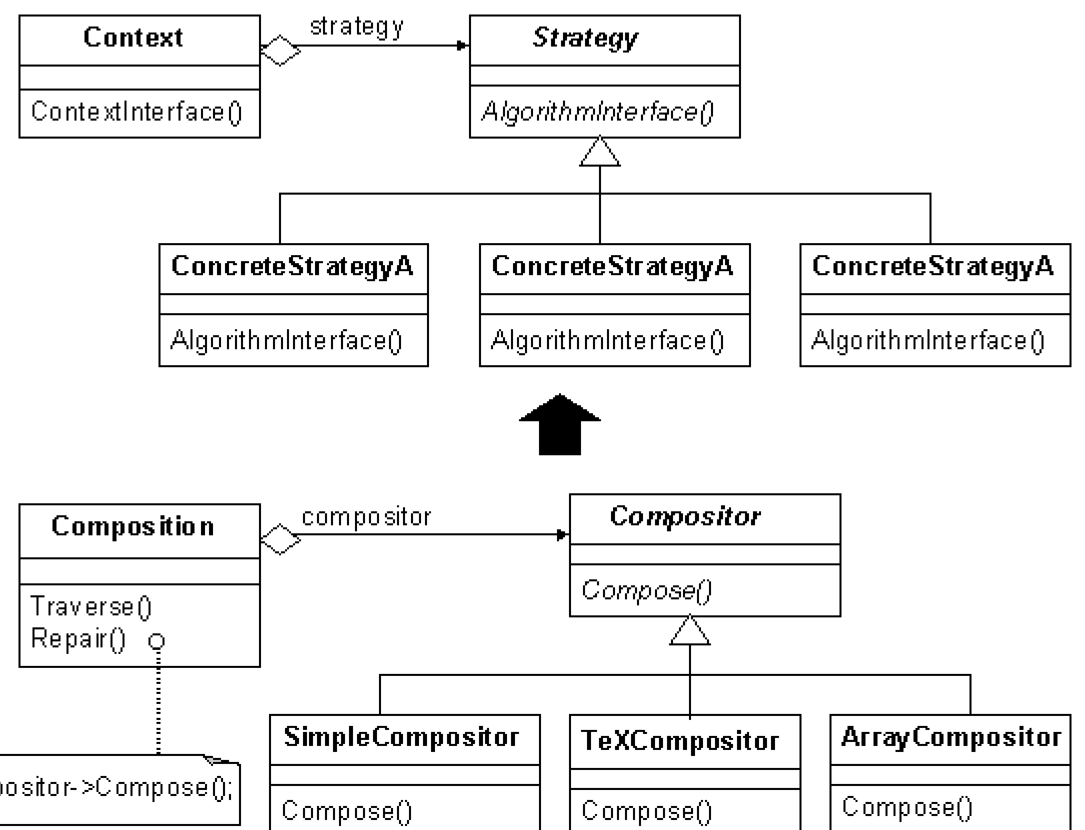


Singleton

- Ensure a class only has one instance, and provide a global point of access to it
  – Many times need only one instance of an object
    - one file system
    - one print spooler
    - …
- Singleton
  – defines a class-scoped instance() operation that lets clients access its unique instance
  – may be responsible for creating its own unique instance



Strategy



- Allow any one of a family of related algorithms to be easily substituted in a system
- Define a family of algorithms, encapsulate each one, and make them interchangeable
- Strategy lets the algorithm vary independently from clients that use it

Further reading

Books

+ Get a good mentor!

- There are many good C++, OOP, UML, OOAD books on the market
  - *The following is just a personal selection*

Introductory C++ book

- S. B. Lippman, J. Lajoie, C++ primer, Addison-Wesley

Reference STL book

- N. Josuttis, The C++ Standard Library, A Tutorial and Reference, Addison-Wesley

More advanced C++ books

- S. Meyers, Effective C++, Addison-Wesley
- S. Meyers, More effective C++, Addison-Wesley
- S. Meyers, Effective STL, Addison-Wesley

UML books

- M. Fowler, UML distilled, Addison-Wesley
- G. Booch et al., The Unified Modeling Language, User Guide, Addison-Wesley

Basic OOAD books

- G. Booch, OO analysis and design, Addison-Wesley
- R. Martin, Designing OO C++ applications using the Booch method, Prentice Hall

Advanced design books

- E. Gamma et al., Design Patterns, Addison-Wesley
- John Lakos, Large-Scale C++ Software Design, Addison-Wesley

Hardcore design book

- A. Alexandrescu, Modern C++ design, Addison-Wesley