Institute of Automation

C++ Basics and Applications in technical Systems

Lecture 9 - Design pattern examples

Institute of Automation
University of Bremen

11th January 2013 / Bremen

WiSe 2012/2013

VAK 01-036

Universität Bremen

Institute of Automation

Overview

Universität Bremen

Lecture schedule

Institute of Automation

Time schedule

- HK **26. Oct.** - Introduction / Simple Program / Datatypes ...
- HK **02. Nov.** - Flow control / User-Defined Data types ...
- CF **09. Nov.** - Simple IO / Functions/ Modular Design ...
- CF **16. Nov.** - C++ Pointer
- CF **23. Nov.** - Object oriented Programming / Constructors
- AL **30. Nov.** - UML / Inheritance / Design principles
- AL **07. Dec.** - Namespace / Operators
- AL **14. Dec.** - Polymorphism / Template Classes / Exceptions
- HK **11. Jan.** - Design pattern examples

Universität Bremen

Important dates

Submission of exercises

1-3 **16. Nov.** - Deadline for submission of Exercise I, 13:00

4-6 **07. Dec.** - Deadline for submission of Exercise II, 13:00

For admission to final exam you need at least 50% of every exercise sheet.

Final project

1-9 **15. Feb.** - Deadline for submission of final project, 13:00

1-9 **19. Feb.** - Final project presentations, 14:00 - 17:00

1-9 **21. Feb.** - Final project presentations, 14:00 - 17:00

Final exam

1-9 **06. Feb.** - Final exam, 10:00-12:00, H3

Virtual methods

Institute of Automation

The keyword `virtual`

Virtual methods of a (base) class, can be reimplemented / redefined in each derived class.

- usage of virtual methods is called run-time polymorphism
- statement: `virtual` `<method declaration>;`

Example

```
class Point
{
public:
  virtual void Move(int iX,
                    int iY);
};
```

Example

```
class Rectangle
  : public Point
{
public:
  void Move(int iX, int iY);
};
```

Bremen

Summarization

Institute of Automation

Remember

- Every class, from which should be inherited, should have a virtual destructor.
- Every class, which has at least one virtual method (incl. the destructor) is called polymorphic type.
- A virtual and overridden method is only accessible from its derivatives. E.g.:

```
void Aircraft::Travel() { Vehicle::Travel(); }
```

- The `virtual` statement is only used in the declaration / header file.
- A method that was declared `virtual` should also be declared `virtual` in all derived children to improve readability.

Templates and polymorphism

Institute of Automation

You already know:

Run-time (dynamic) polymorphism

Polymorphism with virtual methods is called **run-time polymorphism**.

With respect to templates polymorphism is:

Compile-time polymorphism

Polymorphism with templates is called parametric or **compile-time polymorphism**.

Universität Bremen

Code organization with templates

Institute of Automation

Solution:

Separate declaration „Stack.h“ and definition „Stack.cpp“ and inline implementation in separate file „Stack.inl“.

Example

```
// Stack.h
template<class T>
class Stack
{
  void Push(T & value);
};

#include "Stack.inl"
```

Example

```
// Stack.inl
template<class T>
void Stack::Push(T & value)
{
  // ...
}
```

Universität Bremen

Usage of exceptions for error handling

Institute of Automation

Exception handling is mainly used for error handling.

- Code for error handling can be separated from „normal“ code.
- Complexity in code is reduced.

statement for callee

```
throw (<exception object>);
throw (); // no exceptions
```

Without the throw statement any class can be thrown
⇒ unexpected for caller.

statement for caller

```
try {
  // error prone code
}
catch (<exception type>
       [object name])
{
  // error handling
}
```

Universität Bremen

Design Patterns

Institute of Automation

Definition

Design patterns represent generic approaches to fulfill reoccuring requirements. Basic concepts are gathered and reduced to their core functionality. This permits maximal reusability and flexible combination of the patterns.

Notice: The design patterns in this lecture are just a sample. Some patterns have different names depending on the cited source.

Universität Bremen

Institute of Automation

Literature

Definitions and terms in this lecture have partially been taken from [GHJV95]:

GAMMA, E. ; HELM, R. ; JOHNSON, R. ; VLISSIDES, J.:
Design Patterns.
Addison-Wesley Professional, 1995. –
ISBN 0201633612

Universität Bremen

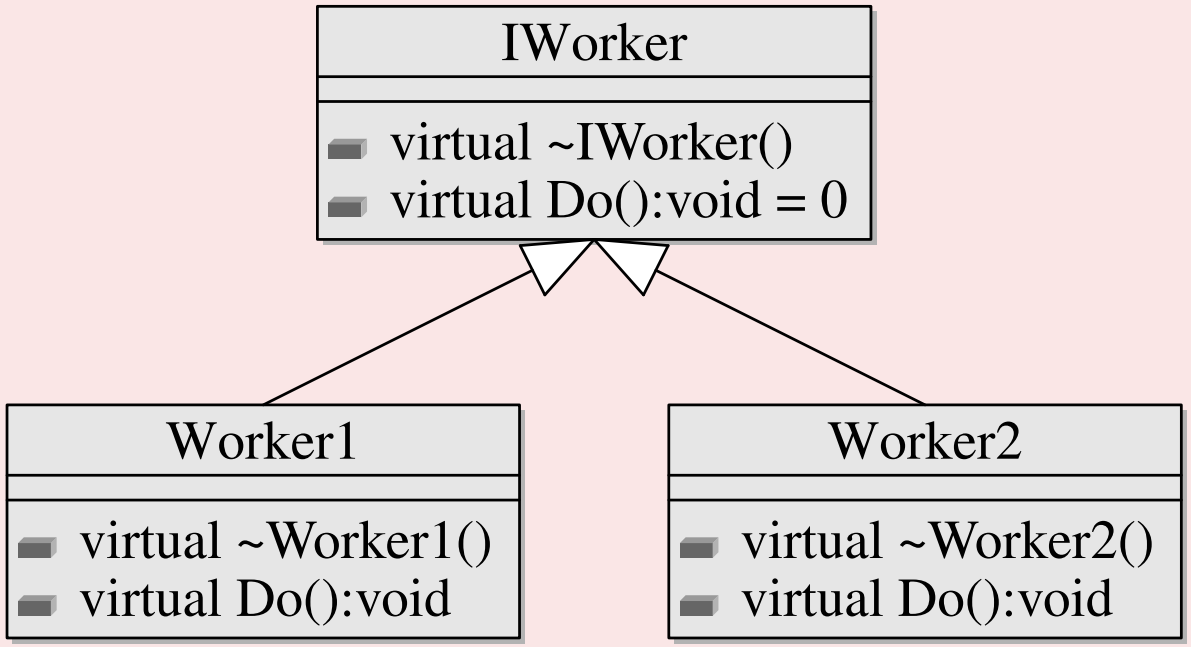
Interface

Purpose:
Defines access to functionality of a class. Different classes can be utilized the same way, when they are derived from the same interface. The user of the interface does not need to care for the actually used class.

Realization:
Create an abstract class with virtual methods to be used as interface. Derrive from the interface class and implement the methods.

Interface

UML



Interface

Institute of Automation

Declaration

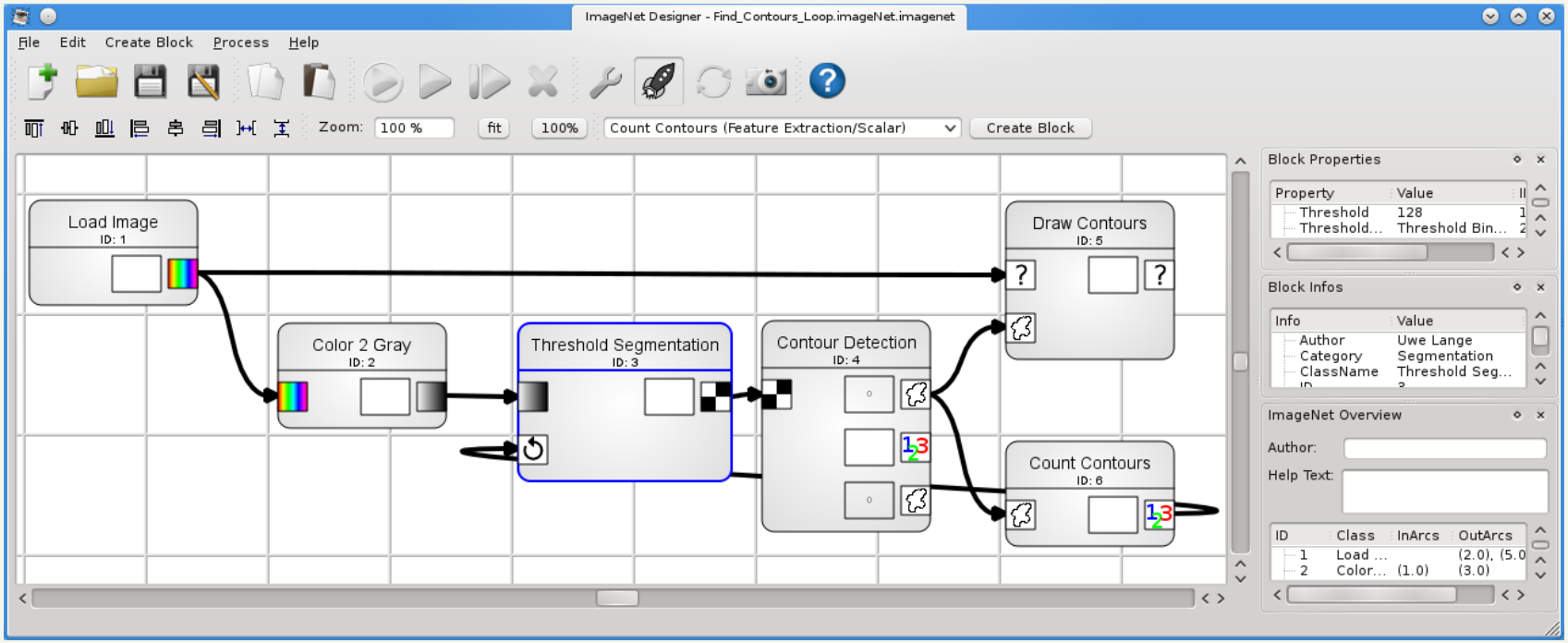
```
class IWorker {
public:
  virtual ~IWorker() ;
  virtual void Do() = 0 ;
}

class Worker1 : public IWorker {
public:
  virtual ~Worker1() ;
  virtual void Do() ;
}

class Worker2 : public IWorker {
public:
  virtual ~Worker2() ;
  virtual void Do() ;
}
```

Definition

```
IWorker::~IWorker() {}

Worker1::~Worker1() {}
void Worker1::Do()
{
  // Do something.
}

Worker2::~Worker2() {}
void Worker2::Do()
{
  // Do something else.
}
```

Universität Bremen

Example: ImageNets



- Graphical algorithms
- Hierarchical modelling
- Parametrizable function blocks
- Extensible via plug-ins
- Embedded documentation
- Directly executable from code

Institute of Automation

Example: ImageNets

Function blocks

- Elementary operations
- In-, out-ports for data
- Configurable properties
- Loaded via plug-ins

Interface

- Uniform handling of all blocks
- Hides actual implementation
- Defines all accessible actions for any block
- Not all actions must be implemented in derrived blocks

Universität Bremen

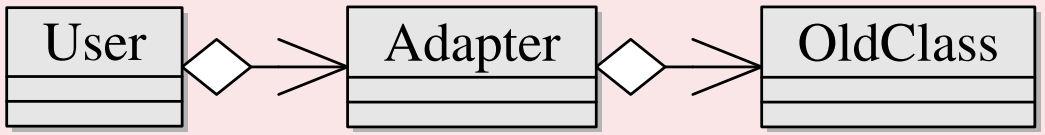
Adapter

Institute of Automation

Purpose:
Adapt an existing class interface to the expectations of a user.

Realization:
Create an adapter class with the required interface and call internally the methods of the old class.

UML



Universität Bremen

Adapter

Institute of Automation

Declaration

```
class OldClass
{
public:
  void Write(
    const char* text);

  OldClass();
  ~OldClass();
}
```

Definition

```
void OldClass::Write(
  const char* text)
{
  cout << text;
}

OldClass::OldClass() {}
OldClass::~OldClass() {}
```

Universität Bremen

Adapter

Institute of Automation

Declaration

```
class Adapter
{
public:
  void Write(
    const string& text);

  Adapter();
  ~Adapter();

private:
  OldClass oldClass;
};
```

Definition

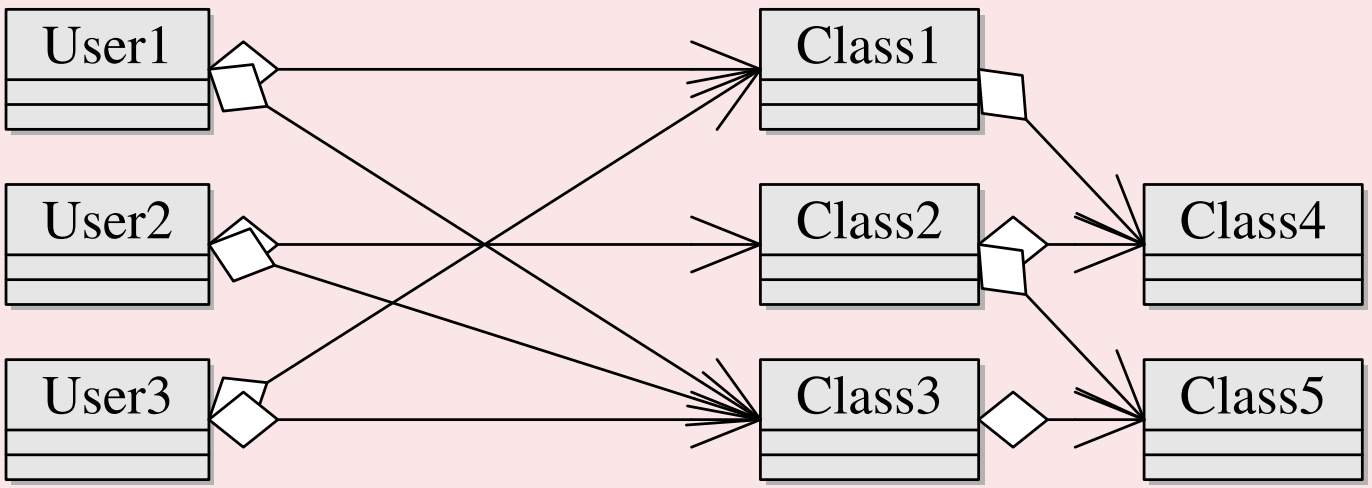
```
void Adapter::Write(
  const string& text)
{
  oldClass.Write(
    text.c_str() );
}

Adapter::Adapter() {}
Adapter::~Adapter() {}
```

Universität Bremen

Facade

Problem: When a set of classes work together to perform a given task, using them can lead to clustered structures.

UML

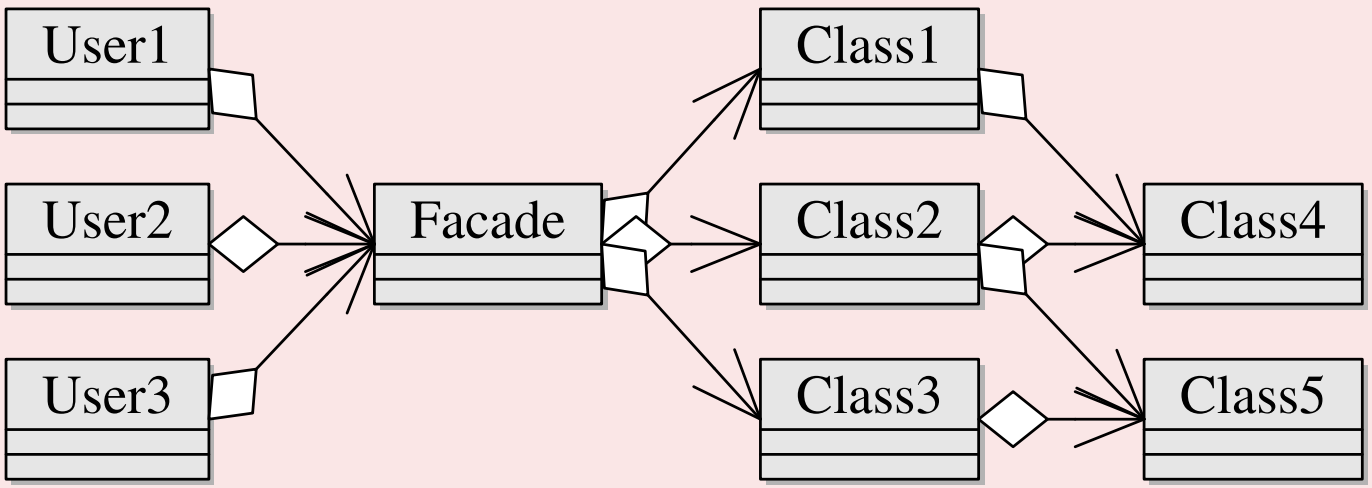


Facade

Institute of Automation

Solution: Use a class as facade to provide one interface to the functionality.

UML



Universität Bremen

Facade

Institute of Automation

Purpose:
Bundle the functionality of a set of classes in one interface.

Realization:
Use the hidden classes in the methods of the facade to realize the functionality. All calls are handled by the facade.

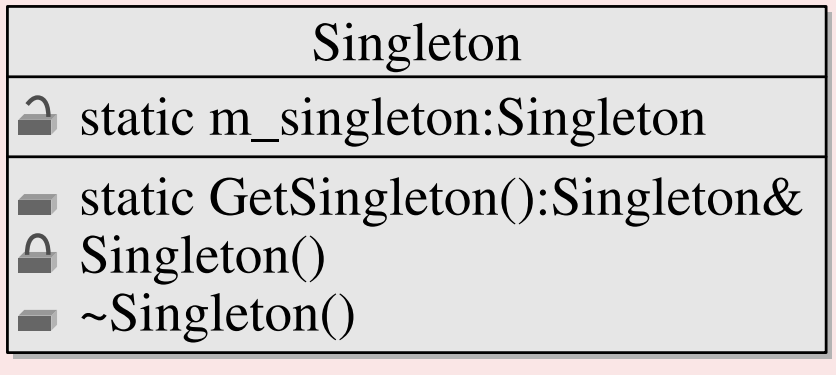
Universität Bremen

Singleton

Institute of Automation

Purpose:
Ensuring that only one object of a certain class exists.

UML



Realization:

- A private constructor forbids construction outside the class.
- The unique object is a static member variable.
- Access to the object only via a static method.
- Singleton is global due to static implementation.

Universität Bremen

Singleton

Institute of Automation

Declaration

```
class Singleton
{
protected:
  static Singleton
    m_Singleton;

public:
  static Singleton&
    GetSingleton();

  ~Singleton();

private:
  Singleton();
};
```

Definition

```
Singleton
    Singleton::m_Singleton;

Singleton& Singleton::
  GetSingleton()
{
  return m_Singleton;
}

Singleton::~Singleton()
{}

Singleton::Singleton()
{}
```

Universität Bremen

Small Exercise

Build a `Logger` as a singleton:

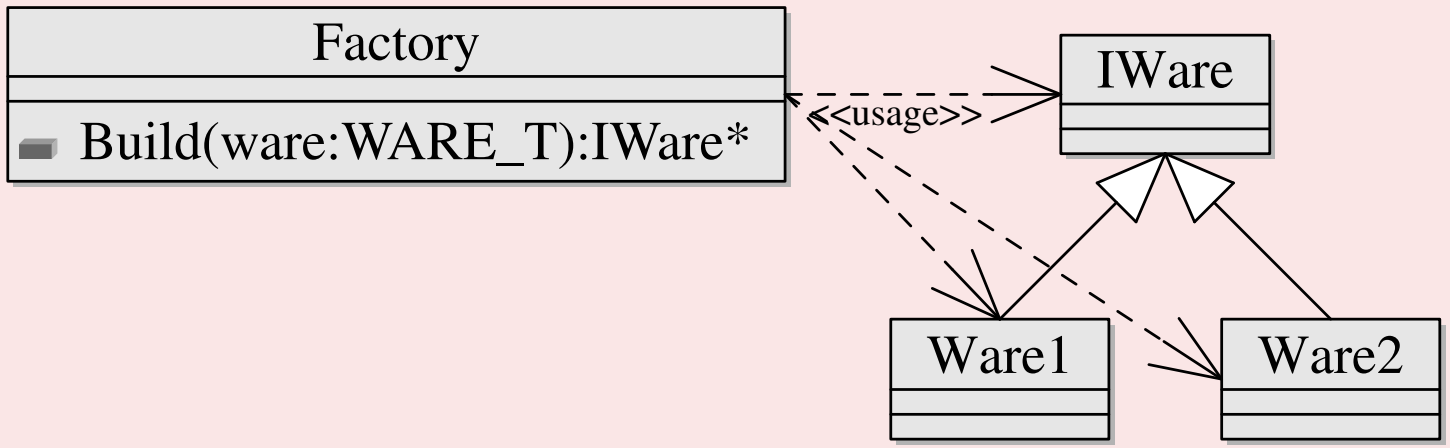
- The `Logger` has an `operator<<` to print a given string to the screen.
- `GetLogger` accesses the static object.
- A `std::string` should be used.
- Test the printing to the screen.
- Try to construct an object of `Logger` outside the class.

Factory

Purpose:
Hide the instanciation of a concrete object for a given interface.

Realization:
Encapsulate the building in a parametrizable method.

UML



Factory

Institute of Automation

Declaration

```
enum WARE_T
  { WARE_1 , WARE_2 } ;

class Factory
{
public:
  IWare* Build(WARE_T ware);

  Factory();
  ~Factory();
};
```

Definition

```
IWare* Factory::
  Build(WARE_T ware)
{
  switch( ware )
  {
    case WARE_1:
      return new Ware1;
    case WARE_2:
      return new Ware2;
  }
  return NULL;
}

Factory::Factory() {}
Factory::~Factory() {}
```

Universität Bremen

Small Exercise

Institute of Automation

Extend the `Logger` from the previous example:

- Create an interface `ILogger` with a virtual `operator<<`.
- Move the cout into a derived class `LoggerScreen`.
- Build a class `LoggerFile` that writes the passed strings into a file.
- The singleton class `Logger` now uses a pointer to `ILogger`.
- The pointer is initialized when first `GetLogger` is called. Therefore this method has now an optional parameter for chosing the output device. A later change is not possible.
- Test both output methods.

Universität Bremen

Example: ImageNets

Function block factory

- Build block from string
- Available blocks from plug-ins
- Only one factory in the program

Singleton

- Prevents creation of more than one instance

Factory

- Building of instances for an interface
- Object type selectable via parameter (e.g. string)

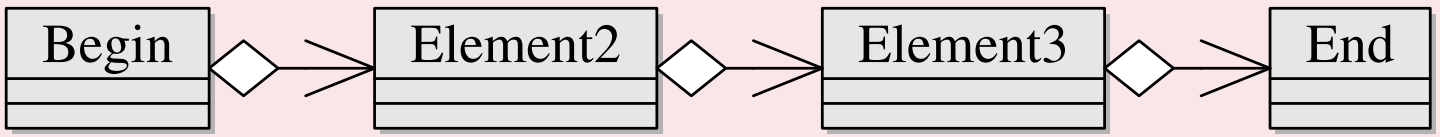
Iterator

Institute of Automation

Purpose:
Process a range of elements one after another. The usage should be the same for different methods to store the elements.

Realization:
Implement a uniform method to linearly traverse arbitrary data sets. The container must supply an element to begin with. After the last data element follows an end element to signal the end of valid data.

UML



Universität Bremen

Iterator

Institute of Automation

Definition

Usage of std container iterators:

- `container.begin()` ⇒ returns first element
- `container.end()` ⇒ returns element behind the last valid data element
- `iterator++` ⇒ goes to the next element
- `*iterator` ⇒ returns the data element the iterator points to

Universität Bremen

Iterator

Institute of Automation

Example

```
int main()
{
  vector< int > intVec;
  intVec.push_back(1);
  intVec.push_back(2);
  intVec.push_back(3);
  for (vector< int >::iterator iter = intVec.begin() ;
    iter != intVec.end() ; iter++)
  {
    cout << *iter << endl;
  }
}
```

Universität Bremen

Iterator

Institute of Automation

Example

```
int main()
{
  list< int > intList;
  intList.push_back(1);
  intList.push_back(2);
  intList.push_back(3);
  for (list< int >::iterator iter = intList.begin() ;
    iter != intList.end() ; iter++)
  {
    cout << *iter << endl;
  }
}
```

Universität Bremen

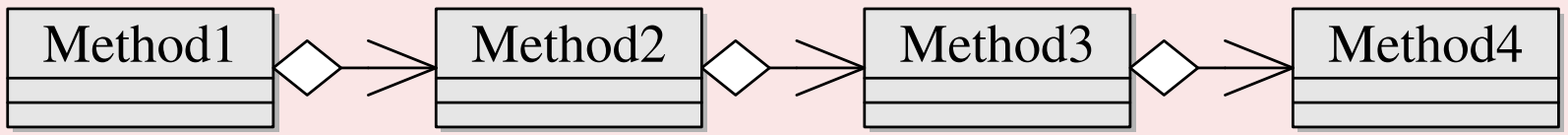
Chain of Responsibility

Institute of Automation

Purpose:
Dynamically handle an event or command.

Realization:
The command is passed along a chain of possible executors or different methods until it is processed.

UML



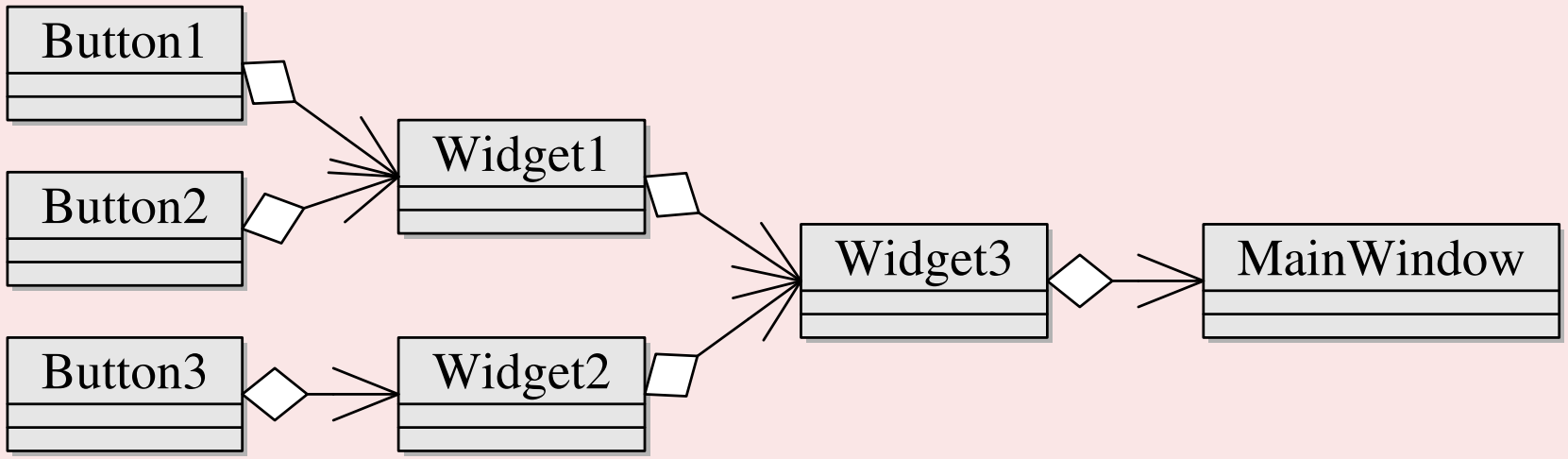
Universität Bremen

Chain of Responsibility

Institute of Automation

One common example of a Chain of Responsibility is the tree structure of a graphical user interface (GUI). A click on a button is passed, when necessary, from the actual element to its parent until the event is processed or discarded.

UML



Universität Bremen

Chain of Responsibility

Institute of Automation

Declaration

```
class Widget {
public:
  void HandleEvent(
    Event* pEvent ) ;

private:
  Widget* m_pParent ;
}
```

Definition

```
void Widget::HandleEvent(
    Event* pEvent )
{
  if ( pEvent->GetName() ==
    "Test" )
  {
    // Handle event.
  }
  else
  {
    // Call next in chain.
    m_pParent->HandleEvent(
      pEvent ) ;
  }
}
```

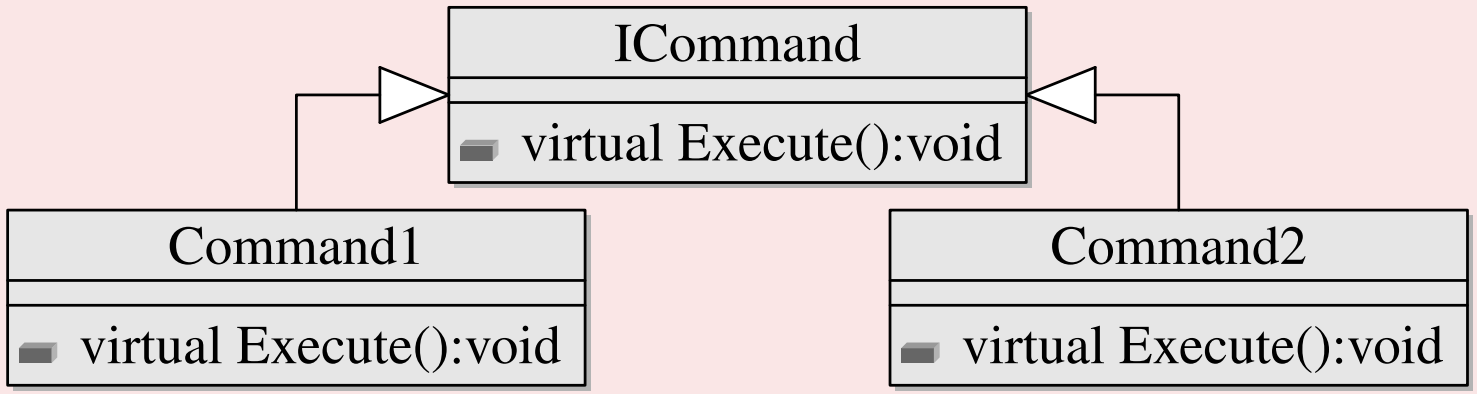
Universität Bremen

Command

Purpose:
Encapsulate a command to be processed.

Realization:
The commands are executed by classes derived from a common interface. These classes hide the execution details.

UML



Institute of Automation

Example

Declaration

```
class ICommand {
public:
  virtual ~ICommand() ;
  virtual string
    GetName() const = 0 ;
  virtual void
    Execute() = 0 ;
}

class Command# :
  public ICommand
{ ... }
```

Declaration

```
class Executor
{
public:
  Executor() ;
  void Execute(
    const string& Name ) ;

private:
  vector< ICommand* >
    m_vpCommands ;
}
```

Universität Bremen

Institute of Automation

Example

Definition

```
Executor::Executor()
{
  // Fills vector with all possible commands.
  m_vpCommands.push_back( new Command1 ) ;
  m_vpCommands.push_back( new Command2 ) ;
  // ...
}

void Executor::Execute( const string& Name )
{
  for ( unsigned int i = 0 ; i < m_vpCommands.size() ; ++i )
  {
    if ( m_vpCommands[ i ]->GetName() == Name )
    {
      m_vpCommands[ i ]->Execute() ;
      return ;
    }
  }
}
```

Bremen

Example: ImageNets

Function block factory

- Build block from string
- Available blocks from plug-ins

Chain of Responsibility

- Series of sub-factories
- One sub-factory for each available block type
- The first sub-factory that can produce the desitred block type is used

Small Exercise

Extend the `Logger` from the previous examples:

- Include both logging methods (screen, file) into `Logger`.
- The actual logging destination is selectable by the user via a `SetDestination` method.
- Reuse the developed classes for the different destinations.
- Test both output destinations.