‘Fortran? C++? Egal!
Ein guter Programmierer kann Spaghetti-Code in jeder Sprache schreiben!’

Wohl nicht ganz ernstgemeinte Bemerkung eines unbekannten Software-Gurus
(Gehört irgendwann Ende der 90er Jahre)

Design Patterns in Object Oriented Design

Dr. Wolfgang F. Mader[1], Peter Steinbach[1]

[1]Institut für Kern- und Teilchenphysik, TU Dresden

Blockkurs Graduiertenkolleg 'Masse, Spektren, Symmetrie', Rathen 2010
10.March 2011

Design Patterns

- **A Design Pattern is:**
  - Description of a Standard Solution for ...
  - ... a Standard Design Problem ...
  - ... in a Certain Context.

- **Goal:**
  - Re-use of Design Information

- **Benefits:**
- Don't Need to Re-invent the Wheel
- Code Structure Easier to Understand
- Easier Maintenance
- Help for Beginners to Learn Good Practices

Design Patterns

- **A Design Pattern is:**
  - Description of a Standard Solution for ...
  - ... a Standard Design Problem ...
  - ... in a Certain Context.

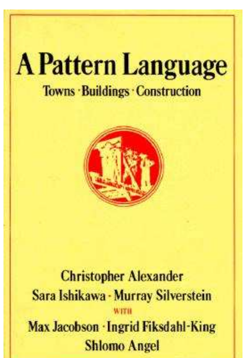
- **Goal:**
  - Re-use of Design Information

- **Benefits:**
- Don't Need to Re-invent the Wheel
- Code Structure Easier to Understand
- Easier Maintenance
- Help for Beginners to Learn Good Practices

Design Patterns

- **A Design Pattern is:**
  - Description of a Standard Solution for ...
  - ... a Standard Design Problem ...
  - ... in a Certain Context.
- **Goal:**
  - Re-use of Design Information
- **Benefits:**
- Don't Need to Re-invent the Wheel
- Code Structure Easier to Understand
- Easier Maintenance
- Help for Beginners to Learn Good Practices

- **Many Analogies Exist:**
  - Song Styles and Theater Plays
  - Novels
  - Architecture and Engineering



- Critical of traditional modern Architecture
- Patterns as solution guides
- Incremental Building
- Empower Laypeople to Create Design

Design Patterns (A Classification)

Erzeugende Muster	Strukturelle Muster	Verhaltensmuster	Weitere Muster
Abstract Factory (Abstrakte Fabrik)	Adapter	Chain of Responsibility (Zuständigkeitskette)	Business Delegate
Builder (Erbauer)	Composite (Kompositum)	Command (Kommando)	Data Access Object
Factory Method (Fabrikmethode)	Bridge (Brücke)	Interpreter	Data Transfer Object (Datentransferobjekt)
Prototype (Prototyp)	Decorator (Dekorierer)	Iterator	Dependency Injection
Singleton (Einzelstück)	Facade (Fassade)	Mediator (Vermittler)	Inversion of Control
	Flyweight (Fliegengewicht)	Memento	Model View Controller
	Proxy (Stellvertreter)	Null Object (Nullobjekt)	Model View Presenter
		Observer (Beobachter)	Plugin
		State (Zustand)	Fluent Interface
		Strategy (Strategie)	
		Template Method (Schablonenmethode)	
		Visitor (Besucher)	
		Interceptor	

Quelle: http://de.wikipedia.org/wiki/Design_Patterns

Design Patterns

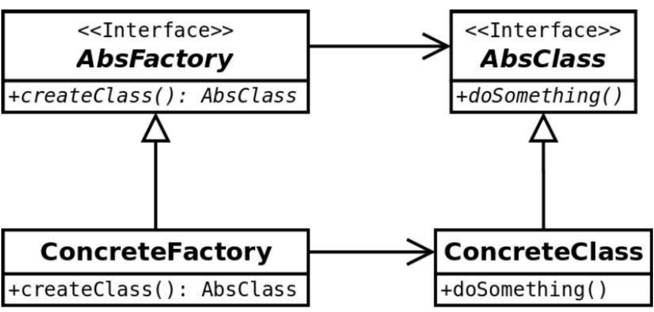
- **Design Patterns Help to Translate Object-Oriented Design Rules**

  - Dependency Management
  - Components
  - Re-Use of Code
  - Ease of Planned (and Unplanned) Changes
  - Maintenance
  - Code Quality

Creational Patterns

- **Organize Object Creation**
- **Class Creational Patterns**
  - Factory Method
  - ...Defer (Part of) Object Creation to Subclasses
- **Object Creational Patters**
  - Abstract Factory
  - Singleton
  - Defer (Part of) Object Creation to Other Objects

(Abstract) Factory Pattern (I)



- Use Case: Portable Software Applications ...
- ....Encapsulate Platform Dependencies (e.g. Windowing System, Databases, ...)
- Provide an Interface to Create Families of Related or Dependent Objects WITHOUT Specifying their Concrete Class
- Easy to Replace Functionality
- Alternative: Prototype

Factory Pattern (II)

```
#include <iostream>

// Produkt
abstract class Mahlzeit {
};

// Konkretes Produkt
class Pizza : public Mahlzeit {
  public:
  Pizza() {
    std::cout << "Pizza gebacken." << std::endl;
  };
};

// Noch ein konkretes Produkt
class Rostwurst : public Mahlzeit {
  public:
  Rostwurst(const char* beilage) {
    std::cout << "Rostwurst gebraten." << std::endl;
    if(beilage) {
      std::cout << "Serviert mit " << beilage << std::endl;
    }
  };
};
```

Factory Pattern (III)

```
// Erzeuger
class Restaurant {
  protected:
  Mahlzeit* _mahlzeit;

  // Die Factory-Methode. Hier wird das konkrete Produkt erzeu
  virtual void bereiteMahlzeitZu() = 0;

  virtual void nimmBestellungAuf() {
    std::cout << "Ihre Bestellung bitte!" << std::endl;
  };

  virtual void serviereMahlzeit() {
    std::cout << "Hier Ihre Mahlzeit. Guten Appetit!" << std::
  };

  public:
  // Diese Methode benutzt die Factory-Methode.
  void liefereMahlzeit() {
    nimmBestellungAuf();
    bereiteMahlzeitZu(); // Aufruf der Factory-Methode
    serviereMahlzeit();
  }
};
```

Factory Pattern (IV)

```
// Konkreter Erzeuger f{\"u}r konkretes Produkt "Pizza"
class Pizzeria : public Restaurant {
  public:
  virtual void bereiteMahlzeitZu() {
    _mahlzeit = new Pizza();
  }
};

// Konkreter Erzeuger f{\"u}r konkretes Produkt "Rostwurst"
class Rostwurstbude : public Restaurant {
  public:
  virtual void bereiteMahlzeitZu() {
    _mahlzeit = new Rostwurst("Pommes und Ketchup");
  }
};

int main() {
  Pizzeria daToni;
  daToni.liefereMahlzeit();

  Rostwurstbude brunosImbiss;
  brunosImbiss.liefereMahlzeit();
}
```

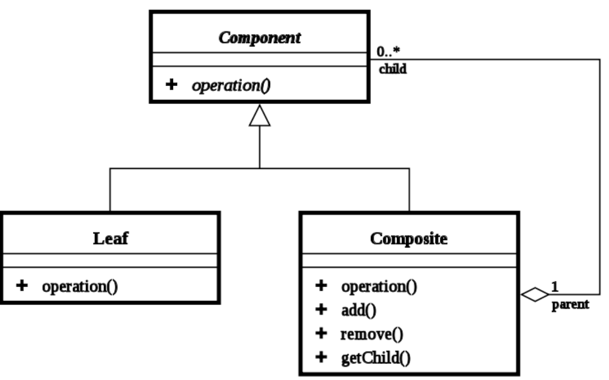
Structural Patterns

- **Compose Complex Structures from Small Ones**

- **Class Structural Patterns**
  - Compose Interfaces or Implementations Using Class Inheritance
  - Adapter Pattern

- **Object Structural Patterns**
  - Compose Objects to Get New Functionality...
  - ... Preferably at Run-Time
  - Adapter, Composite, Decorator, Proxy

Composite Pattern (I)



- Compose objects into tree structures to represent whole-part hierarchies
- Composite lets clients treat individual objects and compositions of objects uniformly
- Recursive composition
- **Example**: Directories contain entries, each of which could be a directory

Composite Pattern (II)

```
#include <iostream>
#include <vector>
using namespace std;

// 2. Create an "interface" (lowest common denominator)
class Component
{
  public:
  virtual void traverse() = 0;
};

class Leaf: public Component
{
  // 1. Scalar class   3. "isa" relationship
  int value;
  public:
  Leaf(int val)
  {
    value = val;
  }
  void traverse()
  {
    cout << value << ' ';
  }
};
```

Composite Pattern (II)

```
class Composite : public Component
{
  // 1. Vector class    3. "isa" relationship
  vector < Component * > children; // 4. "container" coupled t
  public:
  // 4. "container" class coupled to the interface
  void add(Component *ele)
  {
    children.push_back(ele);
  }
  void traverse()
  {
    for (int i = 0; i < children.size(); i++)
    // 5. Use polymorphism to delegate to children
    children[i]->traverse();
  }
};
```

Composite Pattern (III)

```
int main()
{
  Composite containers[4];

  for (int i = 0; i < 4; i++)
    for (int j = 0; j < 3; j++)
      containers[i].add(new Leaf(i *3+j));

  for (i = 1; i < 4; i++)
    containers[0].add(&(containers[i]));

  for (i = 0; i < 4; i++)
  {
    containers[i].traverse();
    cout << endl;
  }
}
```

Output

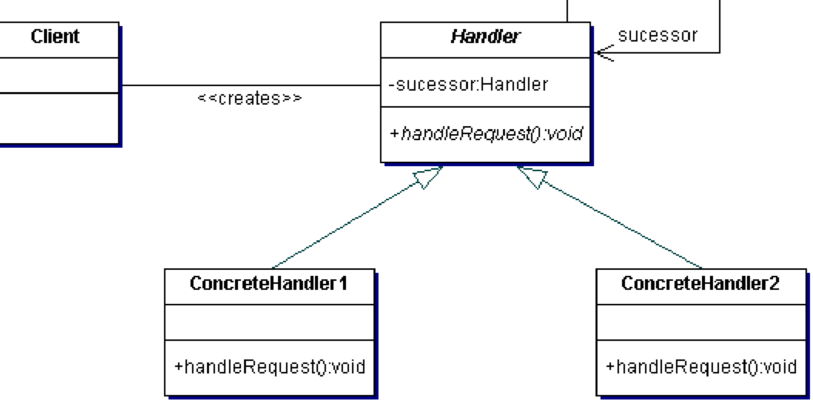
[1 :] main.exe

```
0 1 2 3 4 5 6 7 8 9 10 11
3 4 5
6 7 8
9 10 11
```

Behavioral Patterns

- **Implement Algorithms**
- **Class Behavioral Patterns**
  - Use Inheritance to Separate Algorithm Invariants from Algorithm Variants
  - Template Method
- **Object Behavioral Patters**
  - Use Object Composition to Distribute Algorithm Parts
  - Chain-of-Responsibility, Iterator, State, Observer Strategy

Chain of Responsibility (I)



- Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request
- Chain the receiving objects and pass the request along the chain until an object handles it
- Launch-and-leave requests with a single processing pipeline that contains many possible handlers
- An object-oriented linked list with recursive traversal

Chain of Responsibility (II)

```
class Base
{
  Base *next; // 1. "next" pointer in the base class
  public:
  Base()
  {
    next = 0;
  }
  void setNext(Base *n)
  {
    next = n;
  }
  void add(Base *n)
  {
    if (next)
    next->add(n);
    else
    next = n;
  }
  // 2. The "chain" method in the base class always delegates
  virtual void handle(int i)
  {
    next->handle(i);
  }
};
```

Chain of Responsibility (II)

```
class Handler1: public Base
{
  public:
  /*virtual*/void handle(int i)
  { if (rand() % 3)
    {
      // 3. Don't handle requests 3 times out of 4
      cout << "H1␣passsed␣" << i << "␣␣";
      Base::handle(i); // 3. Delegate to the base class
    }
    else
    cout << "H1␣handled␣" << i << "␣␣";
  } };

class Handler2: public Base
{
  public:
  /*virtual*/void handle(int i)
  { if (rand() % 3)
    {
      cout << "H2␣passsed␣" << i << "␣␣";
      Base::handle(i);
    }
    else
    cout << "H2␣handled␣" << i << "␣␣";
  } };
```

Chain of Responsibility (III)

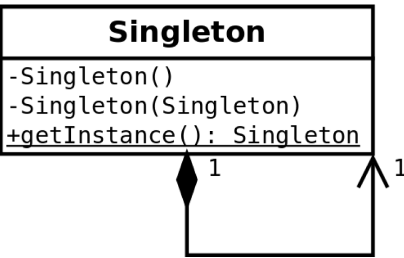
```
class Handler3: public Base
{ public:
  /*virtual*/void handle(int i)
  { if (rand() % 3)
    { cout << "H3 passsed " << i << "  ";
      Base::handle(i);
    }
    else
    cout << "H3 handled " << i << "  "; }
};

int main()
{ srand(time(0));
  Handler1 root;
  Handler2 two;
  Handler3 thr;
  root.add(&two);
  root.add(&thr);
  thr.setNext(&root);
  for (int i = 1; i < 10; i++)
  { root.handle(i);
    cout << '\n'; }
}
```

Output

```
[1:]> main.exe
  H1 passsed 1  H2 passsed 1  H3 passsed 1  H1 passsed 1  H2 handled 1
  H1 handled 2
  H1 handled 3
  H1 passsed 4  H2 passsed 4  H3 handled 4
  H1 passsed 5  H2 handled 5
  H1 passsed 6  H2 passsed 6  H3 passsed 6  H1 handled 6
  H1 passsed 7  H2 passsed 7  H3 passsed 7  H1 passsed 7  H2 handled 7
  H1 handled 8
  H1 passsed 9  H2 passsed 9  H3 handled 9
```

Singleton (I)



- Ensure a class has only one instance, and provide a global point of access to it.
- Application needs one, and only one, instance of an object
  e.g. Histogram Manager
- Private Constructor
- Make the class of the single instance object responsible for creation, initialization, access, and enforcement
- Declare the instance as a private static data member
- Provide a public static member function that encapsulates all initialization code, and provides access to the instance
- The 'static member function accessor' approach will not support subclassing of the Singleton class

Singleton (II)

```
class GlobalClass
{
  int m_value;
  static GlobalClass *s_instance;
  GlobalClass(int v = 0)
  {
    m_value = v;
  }
  public:
  int get_value()
  {
    return m_value;
  }
  void set_value(int v)
  {
    m_value = v;
  }
  static GlobalClass *instance()
  {
    if (!s_instance)
    s_instance = new GlobalClass;
    return s_instance;
  }
};
```

Singleton (III)

```
// Allocating and initializing GlobalClass's
// static data member.  The pointer is being
// allocated - not the object inself.
GlobalClass *GlobalClass::s_instance = 0;

void foo(void)
{
  GlobalClass::instance()->set_value(1);
  cout << GlobalClass::instance()->get_value() << '\n';
}

void bar(void)
{
  GlobalClass::instance()->set_value(2);
  cout << GlobalClass::instance()->get_value() << '\n';
}

int main()
{
  cout << GlobalClass::instance()->get_value() << '\n';
  foo();
  bar();
}
```

Output

```
[1 :] main.exe
        0
        1
        2
```

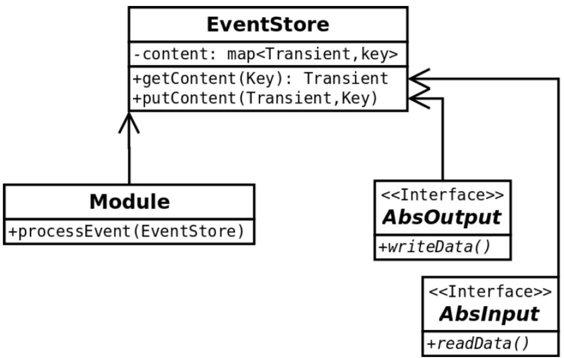
Summary and Outlook

- Design Patterns Useful to Write and Maintain Re-Usable Software Packages

- **Design Patterns:**
  - Structural Patterns
  - Behavioral Patterns
  - Creational Patterns

- **HEP:** Programs Have Sometimes Special Patterns due to Particular Requirements
  - High Throughput
  - Variable Algorithms
  - Long Lifetime of Code
  - Programming Interfaces for Users

Blackboard Pattern



- Model Traditional HEP Data Processing with Objects
- Event Store Holds Event Data
- Processing Module Gets Transient Objects and Puts new Transient Objects
- AbsInput and AbsOutput Decouple IO System from Data Processing
- ATLAS 'StoreGate' BaBar 'event'

Thank you for your Attention!!!!!!!!

References

1) **Design Patterns. Elements of Reusable Object-Oriented Software**
   E. Gamma, R. Helm, R.Johnson, and J. Vlissden
   Addison-Wesley Longman, Amsterdam
2) **Agile Software Development. Principles, Patterns, and Practices**
   R. Marting
   Prentice Hall International
3) **Design Patterns I-III**
   Dr.Stefan Kluth
   Presentation at Workshop on Advanced Software Design, Dresden, 9/2011
4) **Entwurfsmuster**
   http://de.wikipedia.org/wiki/Design_Patterns
5) **Design pattern (computer science)**
   http://en.wikipedia.org/wiki/Design_pattern_(computer_science)
6) **Design Patterns**
   http://sourcemaking.com/design_patterns