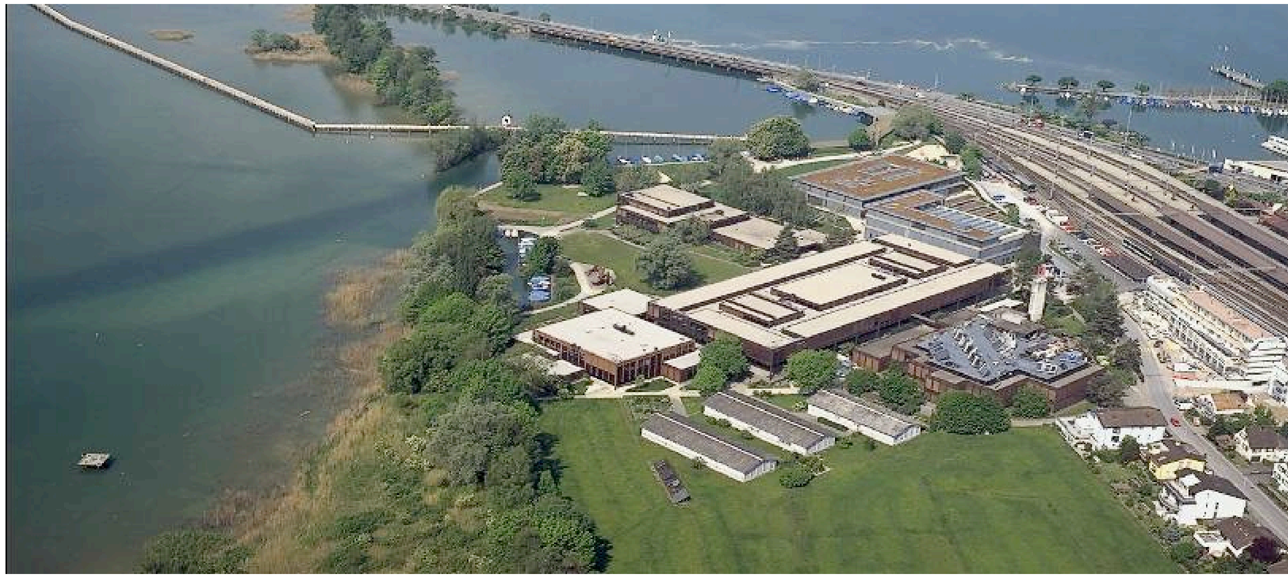
Better Software –
Simpler, Faster

Design Patterns with modern C++

Prof. Peter Sommerlad

HSR - Hochschule für Technik Rapperswil
IFS Institute for Software
Oberseestraße 10, CH-8640 Rapperswil
peter.sommerlad@hsr.ch
http://ifs.hsr.ch
http://wiki.hsr.ch/PeterSommerlad



Peter Sommerlad

peter.sommerlad@hsr.ch

- **Work Areas**
  - Refactoring Tools (C++,Groovy, Ruby, Python) for Eclipse
  - **Decremental Development** (make SW 10% its size!)
  - Modern Software Engineering
  - Patterns
    - Pattern-oriented Software Architecture (POSA)
    - Security Patterns
- **Background**
  - Diplom-Informatiker (Univ. Frankfurt/M)
  - Siemens Corporate Research - Munich
  - itopia corporate information technology, Zurich (Partner)
  - Professor for Software HSR Rapperswil, Head Institute for Software

Credo:

- **People create Software**
  - communication
  - feedback
  - courage
- **Experience through Practice**
  - programming is a trade
  - Patterns encapsulate practical experience
- **Pragmatic Programming**
  - test-driven development
  - automated development
  - **Simplicity: fight complexity**

Goal

INSTITUTE FOR SOFTWARE

- **Sure you can implement Design Patterns as shown in [GoF] in C++**
  - o there are even old C++ code examples
- **But**
  - o standard C++ came after the book and especially templates, STL and std::tr1 (or boost) provide more means to apply the patterns (or some variation)
- **STL even implements some patterns**
  - o e.g. Iterator

GoF Design Patterns

INSTITUTE FOR SOFTWARE

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

object interactions

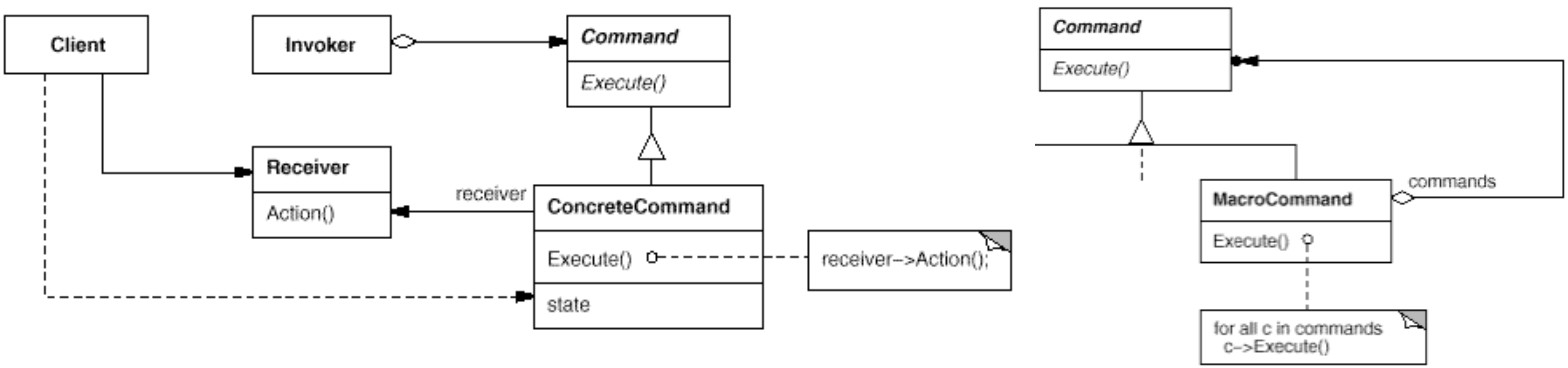
composing objects

remove dependencies on concrete classes when creating objects

Command [GoF]

INSTITUTE FOR SOFTWARE

Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.



Sonntag, 19. April 2009

Command C++

- **C function pointers are the most primitive Command implementation**
  - o but they lack state
- **C++ allows overloading the call operator and can thus implement Functor classes**
  - o instances are callable like functions but can have state
- **std::tr1 or boost define the function<> template to allow storage of arbitrary functor objects (with a given signature)**

➢ if you need more than separate execution the original command pattern might suit you better (>=2 methods)

Example code Command

INSTITUTE FOR SOFTWARE

- **something silly, just for demo purposes**
  - for more, see cute::test

```
#include <iostream>
#include <tr1/functional>
using namespace std::tr1;
struct think {
   void operator()(){
      std::cout << "think " ;
   }
};
struct move {
   void operator()(){
      std::cout << "move " ;
   }
};
struct hide {
   void operator()(){
      std::cout << "hide " ;
   }
};
```

```
void doit(function<void()> f){
   f();
}

void thisIsATest() {
   doit(think());
   doit(move());
   hide()();
   function<void()> f;
   f=move();
   f();
   doit(f);
   f=think();
   f();
}
```

Macro Command: bind+ vector<function<>>

```
struct macroCommand:std::vector<function<void()> >{
   void operator()(){
      std::for_each(begin(),end(),
               bind(&function<void()>::operator(),_1));
   }
};
void testMacroCmd(){
   macroCommand m;
   m.push_back(think());
   m.push_back(move());
   m.push_back(m);
   m();
}
```

Dynamic Polymorphism vs. Policy-based Design

INSTITUTE FOR SOFTWARE

- **“classic” OO technique for variation would be to extract an interface and then provide different implementations of that interface**
- **in C++ this means**
  - run-time overhead of virtual member functions
  - object life-time issues of passed-in parameter objects (must live longer than using objects)
- **often polymorphism is not needed at run-time but only at compile time**
  - when there is no need to vary behavior
  - e.g. for tests, for different platforms, for different usage

Example: Dynamic Polymorphism

INSTITUTE FOR SOFTWARE

```
struct HelloInterface {
    virtual void outputMessage() =0;
    virtual ~HelloInterface(){}
};

struct HelloCoutImpl : HelloInterface {
    void outputMessage() {
        std::cout << "Hello World\n";
    }
};

class HelloWorld {
    HelloInterface &hello;
public:
    HelloWorld(HelloInterface& hello):hello(hello){}
    void operator()(){ // just to demo a functor...
        hello.outputMessage();
    }
};
int main(){
    HelloCoutImpl toCout;
    HelloWorld doit(toCout);
    doit();
}
```

```
struct HelloTestImpl:HelloInterface {
    std::ostringstream os;
    void outputMessage() {
        os << "Hello World\n";
    }
};
void thisIsATest() {
    HelloTestImpl forTest;
    HelloWorld doit(forTest);
    doit();
    ASSERT_EQUAL("Hello World\n",forTest.os.str());
}
```

- **Variation for Test**
- **Extracted Interface**
- **polymorphic call**

Alternative PBD Static Polymorphism

INSTITUTE FOR SOFTWARE

```
struct HelloCoutImpl {
    static void outputMessage() {
        std::cout << "Hello World\n";
    }
};
template <typename HelloInterface>
class HelloWorld {
public:
    void operator()(){ // just to demo a functor...
        HelloInterface::outputMessage();
    }
};
int main(){
    HelloWorld<HelloCoutImpl> doit;
    doit();
}
```

- **Policy-based Design**
- **policy for output variation at compile time**

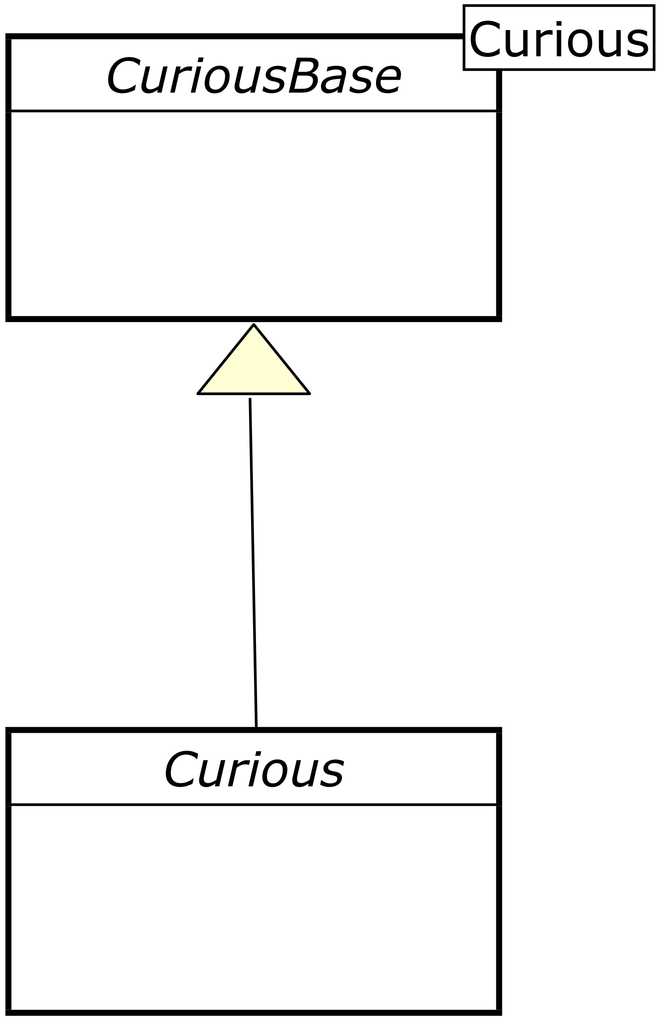
- **Implicit Interface**
- **Variation for Test**

```
struct HelloTestImpl{
    static std::ostringstream os;
    static void outputMessage() {
        os << "Hello World\n";
    }
};
std::ostringstream HelloTestImpl::os;

void thisIsATest() {
    HelloWorld<HelloTestImpl> doit;
    doit();
    ASSERT_EQUAL("Hello World\n",
                 HelloTestImpl::os.str());
}
```

CRTP - curiously recurring template parameter

- **CRTP is when a class inherits from a template class that takes the derived class as template parameter:**

```
template <typename Derived>
class CuriousBase {
  ...
};

class Curious : public CuriousBase<Curious> {
  ...
};
// [VanJos]
```



Singleton (partially) different!

INSTITUTE FOR SOFTWARE

- **Don't use Singleton: Parameterize from Above!**

Ensure a class only has one instance, and provide a global point of access to it. :-(

...

4. Permits a variable number of instances. The pattern makes it easy to change your mind and allow more than one instance of the Singleton class. Moreover, you can use the same approach to ***control the number of instances that the application uses****. Only the operation that grants access to the Singleton instance needs to change.*

- **but for the case given in the 4th consequence we can apply CRTP to count instances for all your classes that need to limit the number of instances**
  - we throw an error if you try to instantiate more!

CRTP limit object count for a class (usage/test)

- **ensure there can be only one one object or two**

```
#include "LimitNofInstances.h"
class one:LimitNofInstances<one,1>{};
class two:LimitNofInstances<two,2>{};
void testOnlyOne() {
   one theOne;
   ASSERT_THROWS(one(),std::logic_error);
}

void testTwoInstances(){
   two first;
   {
      two second;
      ASSERT_THROWS(two(),std::logic_error);
   }
   two nextsecond;
   ASSERT_THROWS(two(),std::logic_error);
}
```

CRTP limit object count for a class (impl)

```
#include <stdexcept>
template <typename TOBELIMITED, unsigned int maxNumberOfInstances>
class LimitNofInstances {
   static unsigned int counter;
protected:
   LimitNofInstances(){
      if (counter == maxNumberOfInstances)
         throw std::logic_error("too many instances");
      ++counter;
   }
   ~LimitNofInstances(){
      --counter;
   }
};
template <typename TOBELIMITED, unsigned int maxNumberOfInstances>
unsigned int
LimitNofInstances<TOBELIMITED,maxNumberOfInstances>::counter(0);
```

Null Object only implicitely in [GoF]

INSTITUTE FOR SOFTWARE

> *A Null Object provides a surrogate for another object that shares the same interface but does nothing. Thus, the Null Object encapsulates the implementation decisions of how to do nothing and hides those details from its collaborators*
>
> *Bobby Woolf in [PLoP3]*

- **requires shared interface and polymorphism**

```
struct HelloInterface {
    virtual void outputMessage() =0;
    virtual ~HelloInterface(){}
};
struct HelloNullImpl:HelloInterface {
    void outputMessage(){}
};
int main(){
    HelloNullImpl noOutput;
    HelloWorld doNothing(noOutput);
    doNothing();
}
```

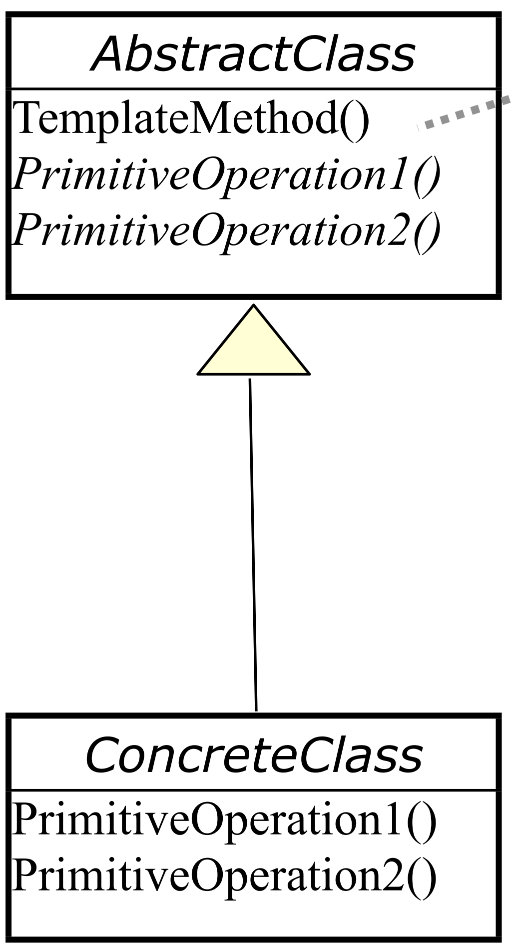
dynamic Polymorphism

```
struct HelloNullImpl{
    static void outputMessage(){}
};
template <typename HelloInterface=HelloNullImpl>
class HelloWorld {
public:
    void operator()(){ // just to demo a functor...
        HelloInterface::outputMessage();
    }
};
int main(){
    HelloWorld<> donothing;
    donothing();
}
```

static Polymorphism

Template Method

Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.

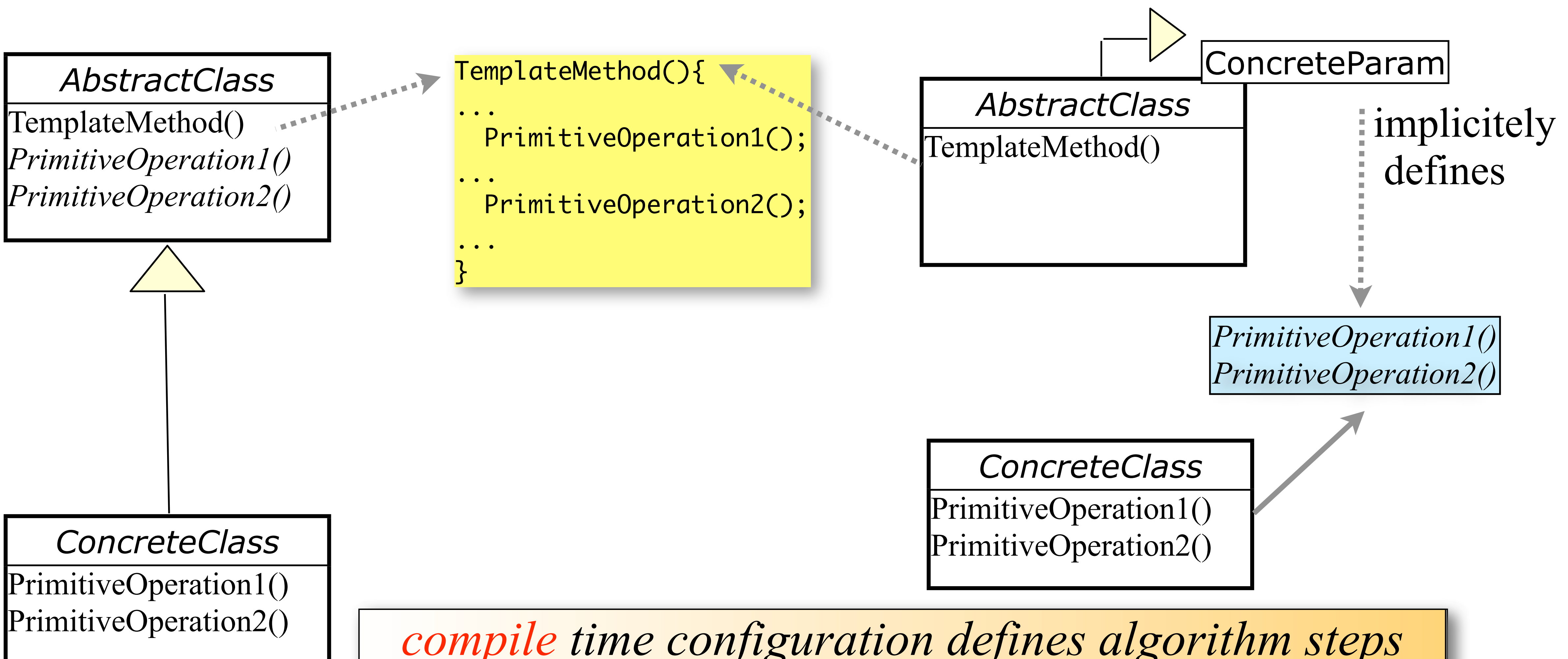



```
TemplateMethod(){
...
  PrimitiveOperation1();
...
  PrimitiveOperation2();
...
}
```

run time configuration defines algorithm steps implementation

Template Method

Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets superclass redefine certain steps of an algorithm without changing the algorithm's structure.



compile time configuration defines algorithm steps implementation

Implementing static Template Method

INSTITUTE FOR SOFTWARE

- write a template class that inherits from its template parameter
- In this class implement the algorithm (Template Method) calling out to Super's methods

```
template <typename Super>
class Base: Super {
  public:
    void algorithm() {
      Super::step1();
      //... something
      Super::step2();
    }
};
```

```
class TMImpl{
  protected:
    void step1(){...}
    void step2(){...}
};

//...
Base<TMImpl> object;
object.algorithm();

Base<AnotherImpl> object2;
object2.algorithm();
```

template class cute::runner

```
template <typename Listener=null_listener>
class runner : Listener{
  bool runit(test const &t){
    try {
      Listener::start(t);
      t(); // run the test
      Listener::success(t,"OK");
      return true;
    } catch (cute::test_failure const &e){
      Listener::failure(t,e);
    } catch (std::exception const &exc){
      Listener::error(t,test::demangle(exc.what()).c_str());
    } catch(...) {
      Listener::error(t,"unknown exception thrown");
    }
    return false;
  }
....
};
```

Null-Object Pattern

Template Method Pattern

Null Object null_listener

```
struct null_listener{ // defines Contract of runner
parameter
	void begin(suite const &s, char const *info){}
	void end(suite const &s, char const *info){}
	void start(test const &t){}
	void success(test const &t,char const *msg){}
	void failure(test const &t,test_failure const &e){}
	void error(test const &t,char const *what){}
};
```

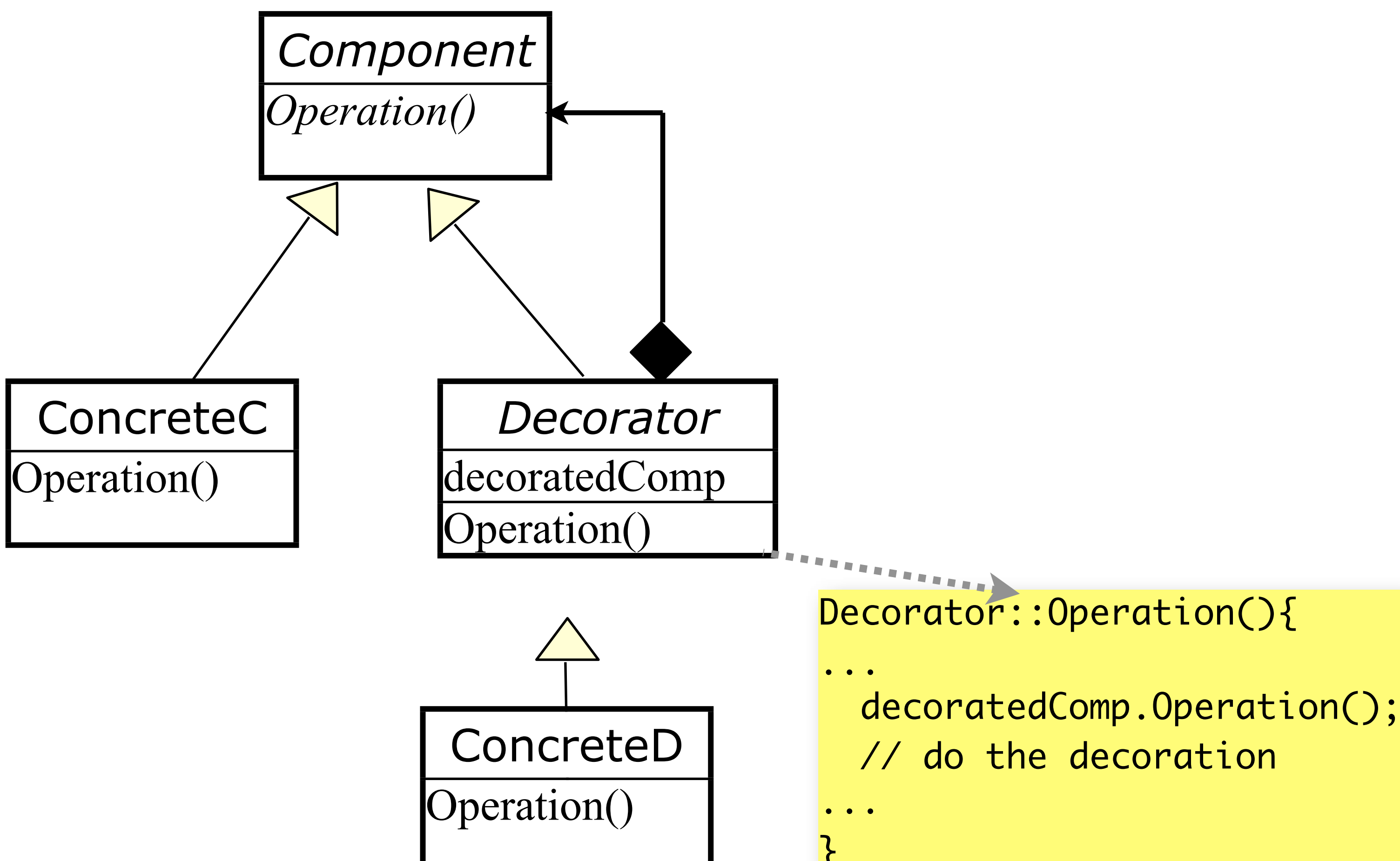
Null-Object Pattern

Template Method Pattern Interface

Decorator

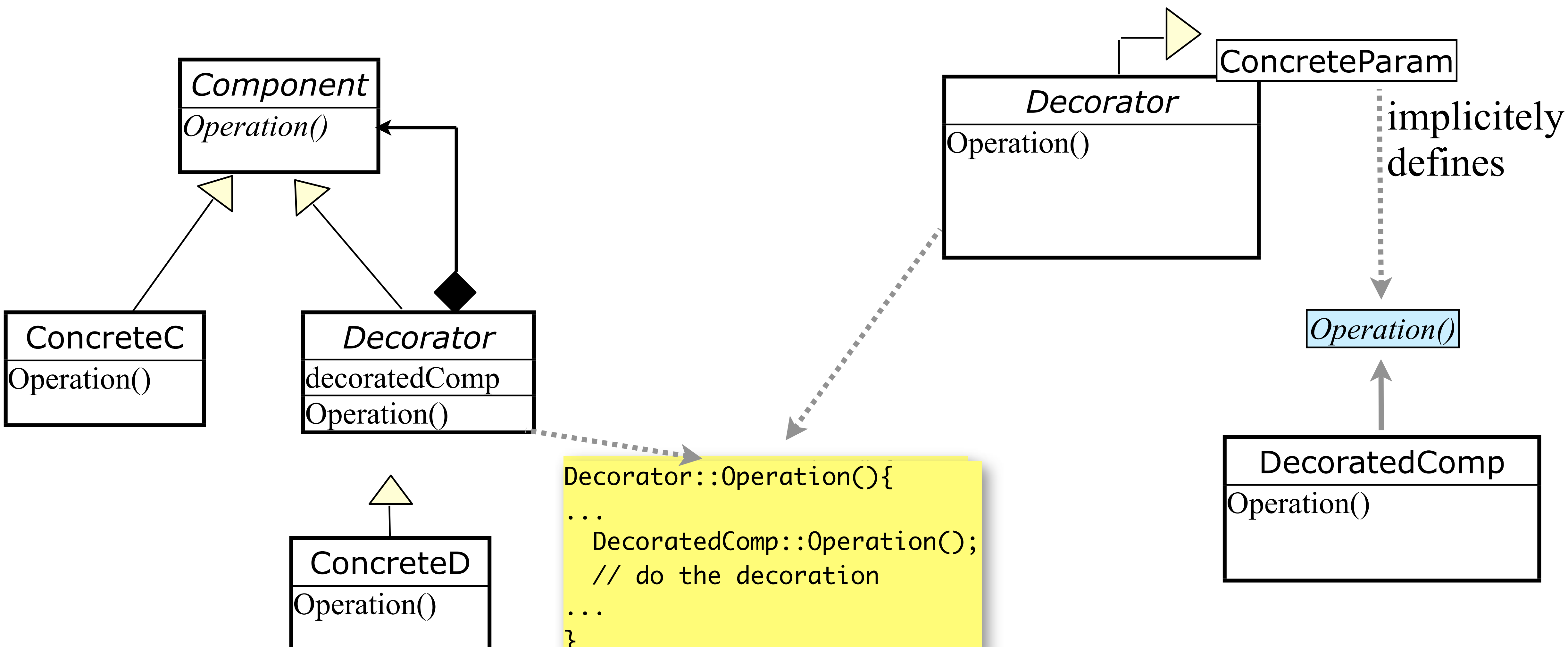
INSTITUTE FOR SOFTWARE

Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.



Decorator

Attach additional responsibilities to an object statically. Decorators provide a flexible alternative via subclassing for extending functionality.



Decorator counting_listener

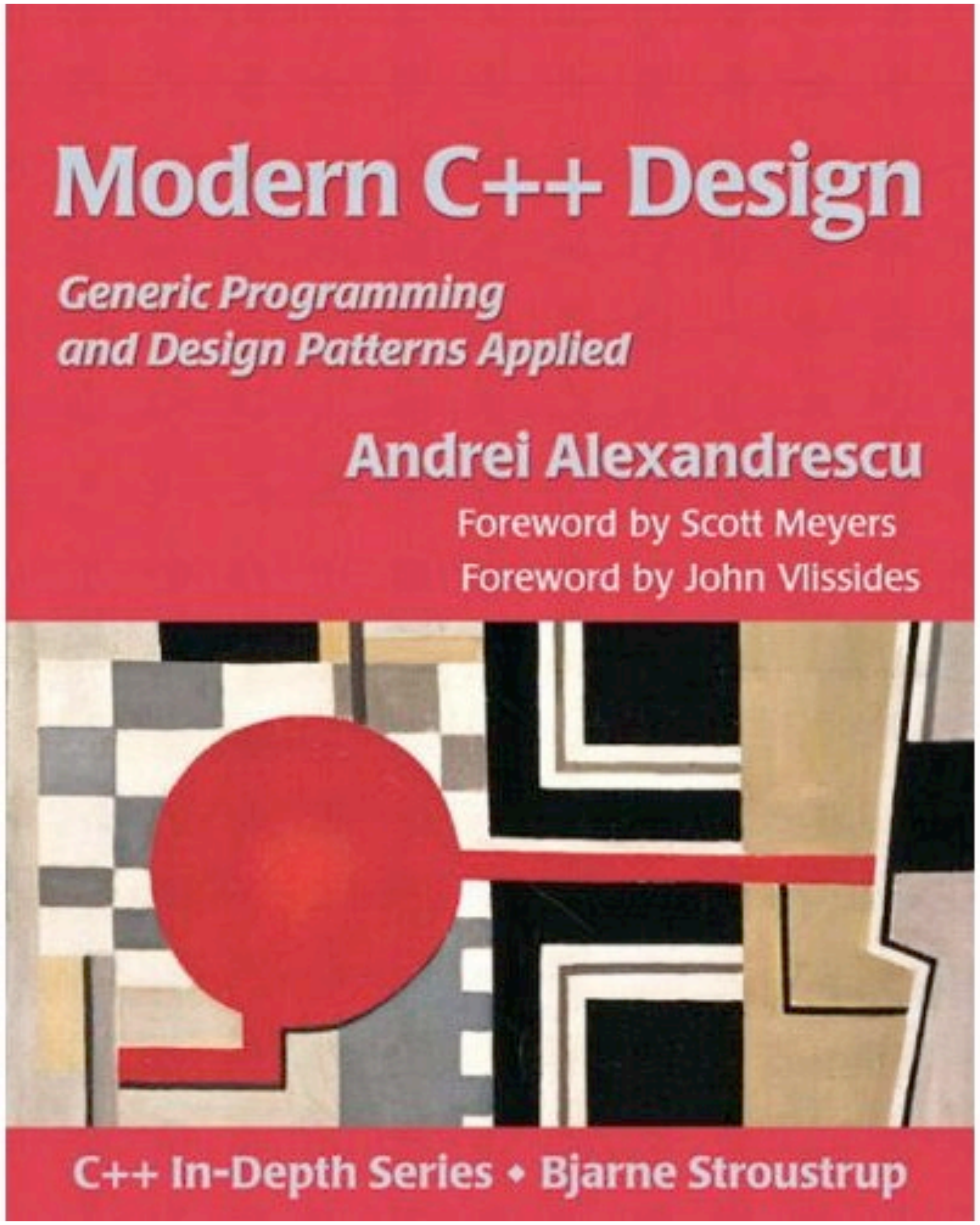
```
template <typename Listener=null_listener>
struct counting_listener:Listener{
  counting_listener() :Listener()
      ,numberOfTests(0),successfulTests(0),failedTests(0){}
      void start(test const &t){
                  ++numberOfTests;
                  Listener::start(t);
      }
      void success(test const &t,char const *msg){
                  ++successfulTests;
                  Listener::success(t,msg);
      }
      void failure(test const &t,test_failure const &e){
                  ++failedTests;
                  Listener::failure(t,e);
      }
...
};
```

Null-Object Pattern

Template Method Pattern Interface

What else?

- **Have a look in Andrej's book:**
  - however, a bit of an overdose...
  - use the stuff from boost or std::tr1 instead of DIY



Outlook: Adapter with Concept Maps

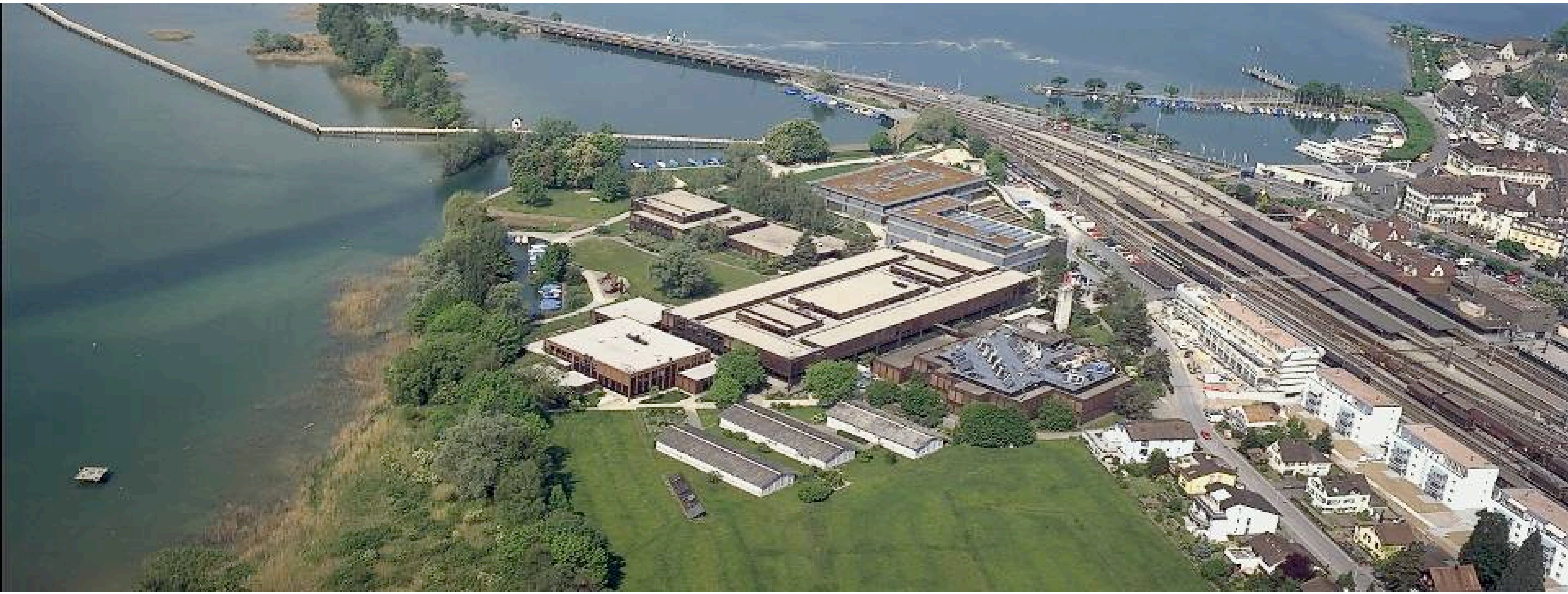
INSTITUTE FOR SOFTWARE

- Concepts and concept maps provide “promising generic, non-intrusive, efficient, and identity preserving adapters.” [JMS07]
- Example: Adapter for arrays to be used in the new for loop syntax with the For<T> concept:

```
template<typename T, size_t N>
concept_map For<T[N]> {
  typedef T* iterator;
  T* begin(T (&array)[N]) {
    return array;
  }
  T* end (T (&array)[N]) {
    return array + N;
  }
}
```

Questions?

INSTITUTE FOR SOFTWARE



- Or contact me at **peter.sommerlad@hsr.ch**

References

- **[GoF]**
  - Gamma E., Helm R., Johnson R., Vlissides J.: Design Patterns - Elements of reusable object-oriented Design, AW 1994
- **[JMS07]**
  - Jaakko Järvi, Matthew A. Marcus, and Jacob N. Smith. Library composition and adaptation using c++ concepts. In GPCE '07: Proceedings of the 6th international conference on Generative programming and component engineering, pages 73–82, New York, NY, USA, 2007. ACM.
- **[VanJos]**
  - David Vandervoorde, Nicolai Josuttis: C++ Templates: The Complete Guide
- **[PLoP3]**
  - Martin R., Riehle D., Buschmann F.: Pattern Languages of Program Design 3, AW 1998