(Ein wenig) Advanced C++

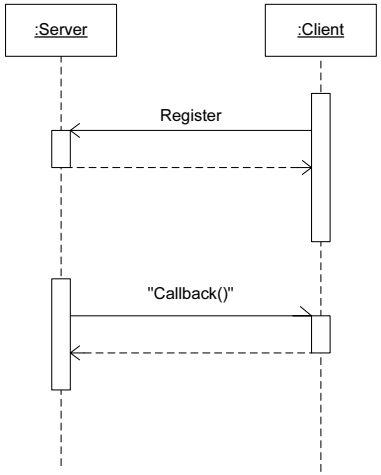
Flexible Adapter mit Templates und Funktionszeigern

Daniel Lohmann

Lehrstuhl für Informatik 4
Verteilte Systeme und Betriebssysteme

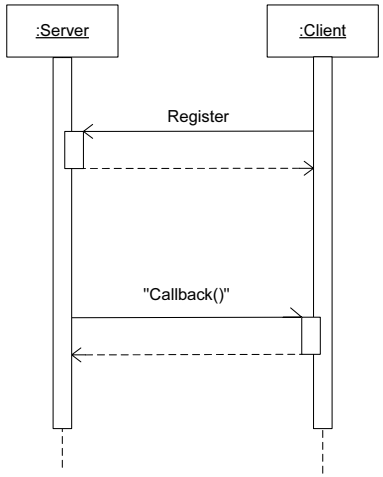
http://www4.informatik.uni-erlangen.de/~lohmann
daniel.lohmann@informatik.uni-erlangen.de

Motivation: Das Rückrufproblem (I)



synchron

Motivation: Das Rückrufproblem (II)



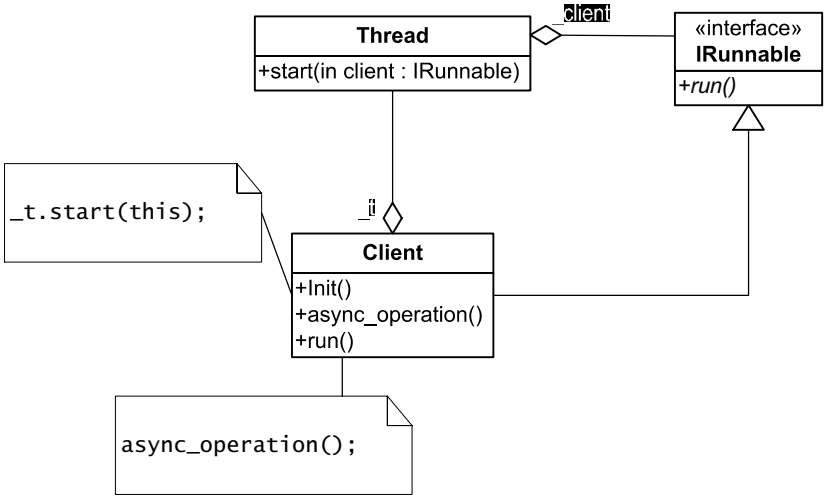
asynchron

Realisierung von Rückrufen

„Wie ruft der Server zurück?“

- Der prozedurale Ansatz (C): Rückruffunktion (Callback)
  - Client übergibt Server Zeiger auf eine Funktion
  - Funktion muss einer bestimmten Signatur entsprechen
  - Server ruft Client-Funktion über Funktionszeiger auf
- Der OO-Ansatz (Java): Interface
  - Client übergibt Server Zeiger auf Objektinstanz (this)
  - Client-Klasse muss von bestimmtem Interface erben
  - Server ruft Client über das Interface auf

Erzeugen eines Threads (OO-Ansatz)



Implementierung (I)

```
struct IRunnable {
  virtual void run() = 0;
};
class Thread {
  IRunnable* _client;
public:
  Thread()
    : _client( 0 ) {}

  void start( IRunnable& client ){
    // ... create new thread
    // starting in _client.run()
  }
};
```

```
class Client : public IRunnable {
  Thread _t;
public:
  virtual void async_worker() {
    // arbeite
    // ...
  }
  Client(){}

  void init() {
    _t.start( *this );
  }
  virtual void run() {
    async_worker();
  }
};
```

OO-Ansatz

Implementierung (II)

```
struct IRunnable {
  virtual void run() = 0;
};
class Thread {
  IRunnable* _client;
public:
  Thread()
    : _client( 0 ) {}

  void start( IRunnable& client ){
    // ... create new thread
    // starting in _client.run()
  }
};
```

```
class Client : public IRunnable {
  Thread _t;
public:
  virtual void async_worker() {
    // arbeite
    // ...
  }
  Client(){}

  void init() {
    _t.start( *this );
  }
  virtual void run() {
    async_worker();
  }
};
```

OO-Ansatz

```
typedef ... thread_t;

typedef void (*PFTHREAD)(void*);
thread_t create_thread( PFTHREAD run,
                        void* param ) {
  thread_t t;
  // ... create a new thread
  //     starting in run( param )

  return t;
}
```

```
namespace Client {
  thread_t _t;

  void async_worker( void* param ) {
    // arbeite
    // ...
  }

  void init() {
    _t = create_thread(Client::async_worker, 0);
  }
}
```

prozeduraler Ansatz

Probleme der „OO-typischen“ Lösung

- Interface-Ansatz hat Nebenwirkungen
  - Implementierungsdetails werden nach außen getragen
  - Signatur und Name der Startmethode sind festgelegt
- Interface-Ansatz skaliert nicht
  - Man kann nur einmal von einem Interface erben
- Die prozedurale Lösung (Funktionszeiger) hat diese Probleme nicht!
  - Warum?

Problem: Mehr als ein Thread

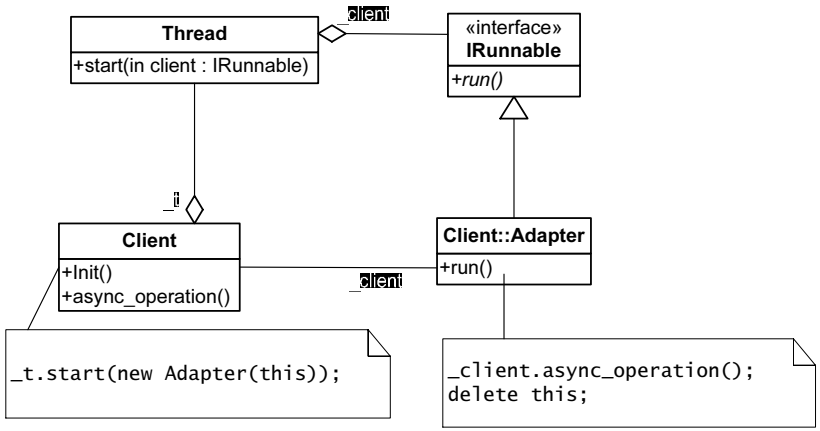
```
class Client : public IRunnable
{
  Thread _t, _t2;
public:
  virtual void async_worker() {
    // arbeite
    // ...
  }
  virtual void async_worker2() {
    // arbeite noch mehr
    // ...
  }

  Client(){}

  void init() {
    _t.start( *this );
    _t2.start( ??? );
  }
  virtual void run() {
    async_worker();
  }
};
```

OO-Ansatz ???

```
namespace Client {
  thread_t _t, _t2;

  void async_worker( void* param ) {
    // arbeite
    // ...
  }

  void async_worker2( void* param ) {
    // arbeite noch mehr
    // ...
  }

  void init() {
    _t = create_thread(
         Client::async_worker, 0);

    _t2 = create_thread(
         Client::async_worker2, 0);
  }
}
```

prozeduraler Ansatz

Idee: Adapter und Delegation



Implementierung mit Adapter

```
class Client {
  Thread _t;
  struct Adapter : public IRunnable {
    Client& _client;
    Adapter(Client& target)
      : _client( target ) {}
    virtual void run() {
      _client.async_worker();
      delete this;
    }
  };
public:
  virtual void async_worker() {
    // arbeite
    // ...
  }
  Client() {}
  void init() {
    _t.start( *new Adapter( *this ) );
  }
};
```

OO-Ansatz mit Delegation und Adapter-Objekt

Bewertung des Adapter-Ansatzes

- Interface-Ansatz hat Nebenwirkungen
  - Implementierungsdetails werden nach außen getragen
  - Signatur und Name der Startmethode sind festgelegt
- Adapter-Ansatz skaliert (ist jedoch umständlich)
  - Für jeden Thread eine eigene Adapter-Klasse
  - Die prozedurale Lösung skaliert immer noch besser
- Ziel: Eine wiederverwendbare Adapterklasse

Problem: Hardcodierter Methodenname

```
class Client {
  Thread _t;
  struct Adapter : public IRunnable {
    Client& _client;
    Adapter(Client& target)
      : _client( target ) {}
    virtual void run() {
      _client.async_worker();
      delete this;
    }
  };
  virtual void async_worker() {
    // arbeite
    // ...
  }
public:
  Client() {}
  void init() {
    _t.start( *new Adapter( *this ) );
  }
};
```

Verbesserung: Methodenzeiger

- Verwendung von Methodenzeigern
  - Im Gegensatz zu Java gibt es in C++ Methodenzeiger
  - Ein Methodenzeiger ist ein erweiterter Funktionszeiger
    - Klasse gehört zur Signatur
    - Aufruf mit Objektinstanz
  - Vorteile wie bei Funktionszeiger
    - Nur noch die Signatur ist festgelegt, nicht mehr der Name
- C# hat das Methodenzeigerkonzept nochmals erweitert
  - sogenannte *Delegates*
  - eigener Datentyp mit speziellen Operationen

Syntax von Funktionszeigern

```
struct B{

  int foo( double data );

  static int bar(double data);
};

void main()
{
  typedef int (*PFBAR)(double);       // Type of pointer to B::bar()
  typedef int (B::*PFFOO)(double);    // Type of pointer to B::foo()
  PFBAR Bar = &B::bar;
  PFFOO Foo = &B::foo;

  int n1 = Bar( 47.11 );

  B Obj;
  B* pObj = &Obj
  int n2 = (Obj.*Foo) ( 47.11 );
  int n3 = (pObj->*Foo) ( 47.11 );

}
```

Adapter mit Methodenzeigern

```
class Client {
  Thread _t, _t2;

  struct Adapter : public IRunnable {
    typedef void ( Client::*PFRUN )();
    Client& _client;
    PFRUN _pfRun;
    Adapter( Client& target, PFRUN pfRun )
      : _client( target ), _pfRun( pfRun ) {}

    virtual void run() {
      (_client.*pfRun)();
      delete this;
    }
  };

  virtual void async_worker() {
    // arbeite
  }
  virtual void async_worker2() {
    // arbeite noch mehr
  }

public:
  Client() {}
  void init() {
    _t.start( *new Adapter( *this, &Client::async_worker ) );
    _t2.start( *new Adapter( *this, &Client::async_worker2 ) );
  }
};
```

Bewertung: Adapter mit Methodenzeigern

- Interface-Ansatz hat Nebenwirkungen
  - Implementierungsdetails werden nach außen getragen
  - Signatur und Name der Startmethode sind festgelegt
- Adapter-Ansatz skaliert nun besser
  - Für jeden Thread eine eigene Adapter-Klasse
  - Aber: Für jede Client-Klasse eine eigene Adapter-Klasse
  - Die prozedurale Lösung skaliert immer noch besser
- Ziel: Adapter unabhängig von der Client-Klasse

➔ Templates

Client-unabhängiger Adapter

```
template< class C >
struct Adapter : public IRunnable {
  typedef void ( C::*PFRUN )();
  C& _client;
  PFRUN _pfRun;
  Adapter( C& target, PFRUN pfRun )
    : _client( target ), _pfRun( pfRun ) {}

  virtual void run() {
    (_client.*pfRun)();
    delete this;
  }
};

class Client {
  Thread _t;
public:
  virtual void async_worker() { //... }
  void init() {
    _t.start( *new Adapter<Client>( *this, &Client::async_worker ) );
  }
};

class AnotherClient {
  Thread _t;
public:
  virtual void mythread() { //... }
  void init() {
    _t.start( *new Adapter<AnotherClient>( *this, &Client::mythread) );
  }
};
```

Bewertung: Client-unabhängiger Adapter

- Interface-Ansatz hat Nebenwirkungen
  - Implementierungsdetails werden nach außen getragen
  - Signatur und Name der Startmethode sind festgelegt
- Adapter-Ansatz skaliert
  - Für jeden Thread eine eigene Adapter-Klasse
  - Aber: Für jede Client-Klasse eine eigene Adapter-Klasse
  - Die prozedurale Lösung skaliert immer noch besser
- Weitere Idee: Adapter in die Thread-Klasse integrieren

Adapter in der Threadklasse

```
class Client {
  XThread _t, _t2;
  virtual void async_worker() {
    // arbeite
    // ...
  }
  virtual void async_worker2() {
    // arbeite noch mehr
    // ...
  }
  Client() {}
public:

  void Init() {
    _t.start( *this, &Client::async_worker );
    _t2.start( *this, &Client::async_worker2);
  }
};
```

XThread

```
class XThread : public Thread {
  template< class C >
  struct Adapter : public IRunnable {
    typedef void ( C::*PFRUN )();
    C& _client;
    PFRUN _pfRun;
    Adapter( C& target, PFRUN pfRun )
      : _client( target ), _pfRun( pfRun ) {}
    virtual void run() {
      (_client.*pfRun)();
      delete this;
    }
  };

public:
  template< class C >
  void start( C& that, void (C::*pfMethod)() ) {
    Thread::start( *new Adapter<C>( *this, pfMethod ) );
  }
};
```