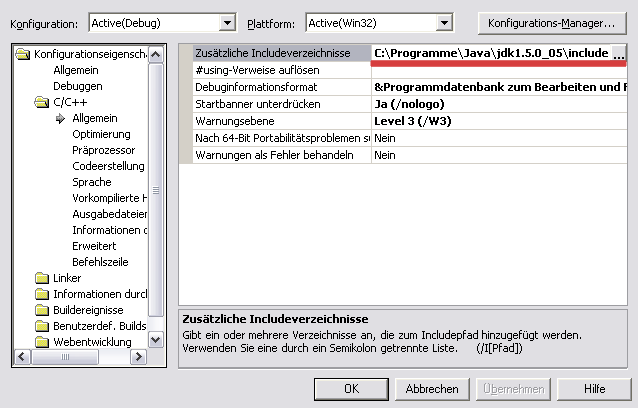
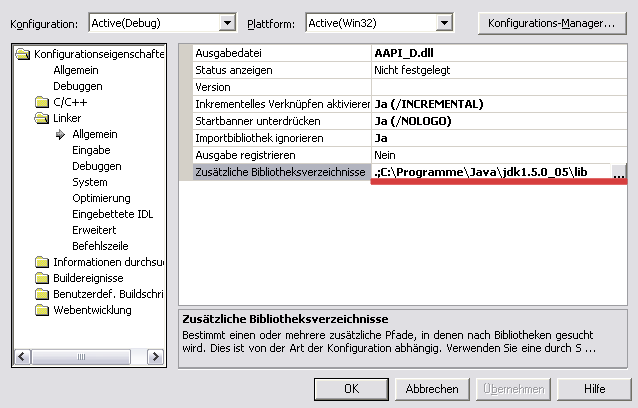
# Vorbereiten der Programmierumgebung

## Erstellen einer API-DLL für AIMSUN mit Visual C++

Diese API-DLL soll über das Java Native Interface (JNI) eine Java Virtual Machine (JVM) starten.

## Visual Studio

Zunächst sind die Verzeichnisse $(JAVA\_HOME)\include und $(JAVA\_HOME)\include\win32 des JDK in den Eigenschaften des Projekts unter Konfigurationseigenschaften🡪 C/C++🡪 Allgemein🡪 Zusätzliche Includeverzeichnisse einzutragen.

Gleiches gilt im Bereich Linker für das Bibliotheksverzeichnis $(JAVA\_HOME)\lib, damit die jvm.lib mit gelinkt werden kann. Dabei ist das aktuelle Projektverzeichnis („.“, ein einzelner Punkt) auch mit aufzunehmen, da dort eine Bibliothek mit AIMSUN-Funktionen zu finden ist.

Als erster Schritt wird über diese DLL nun eine JVM gestartet. Dafür wird aus der Registry ausgelesen, welches die installierte Java-Version ist und wo diese zu finden ist. Es wird geprüft, ob die vorhandene Version mindestens der geforderten entspricht. Die Versionsangabe für die Mindestanforderungen erfolgt über den Präprozessor, einstellbar unter Konfigurationseigenschaften🡪 C/C++🡪 Präprozessor🡪 Präprozessordefinitionen (JDK\_MAJOR\_VERSION und JDK\_MINOR\_VERSION mit den Werten \“1\“ und \“6\“ (die gequoteten Anführungszeichen sind wichtig!)).

### Voraussetzungen: Verzeichnisstruktur für Java-Klassen

Über diese DLL soll Java-Code gestartet werden. Die class-Dateien werden in einem Verzeichnis *java* erwartet, das im gleichen Verzeichnis liegen muss wie das in AIMSUN verwendete ANG-Modell. Hier muss die entsprechende Ordnerhierarchie liegen, z. B. de\dfg\oc\otc\manager mit den class-Dateien.

Setzen Sie in den Eigenschaften des Java-Projekts das Ausgabeverzeichnis (Build path) direkt auf dieses Verzeichnis, so muss nichts mehr kopiert oder verschoben werden.

### Synchronisierung: Windows-Events

Zur ersten Synchronisierung zwischen AIMSUN-API und Java-Code werden Windows-Events verwendet. Um diese auslösen zu können, benötigt der Java-Code eine DLL für die zugehörige native Funktion. Diese kann direkt aus dem Visual Studio Projekt erzeugt werden – ist sie nicht vorhanden, kann der Java-Teil nicht gestartet werden. Für die Erzeugung dieser DLL gibt es im Visual Studio Projekt die Konfiguration JavaEventManager.



Nach einem Wechsel der Konfiguration sollte der nächstfolgende Compilerlauf immer mit Erstellen🡪Neu erstellen gestartet werden. Die erzeugte DLL muss den Namen JavaNativeEvent.dll tragen und im gleichen Verzeichnis wie die Class-Dateien liegen. Visual Studio ist schon so eingestellt (Linker🡪Allgemein🡪Ausgabedatei).

### Vom Quelltext zum lauffähigen Projekt

Da nur Quelltexte vorliegen, müssen zunächst alle Komponenten kompiliert werden.

Zunächst die DLL für AIMSUN (API, Konfiguration *Debug* oder *Release*) und diejenige für die Java-GUI (JavaEventManager), dann die Java-GUI.

Anschließend muss im AIMSUN-Modell auf die richtige DLL (Debug/Release) verwiesen werden: Scenario🡪Properties🡪AimsunAPI.

### Java Heap-Speicher

Man kann der Java-Laufzeitumgebung mit Hilfe von Kommandozeilenparametern mitteilen, wie viel Heap-Speicher reserviert werden soll:

java -Xms<initial heap size> -Xmx<maximum heap size>

Im OTC-Projekt wird die Java-Laufzeitumgebung jedoch aus nicht über die Kommandozeile gestartet, sondern aus der API-DLL heraus. Um den Heap-Speicher anzupassen, müssen daher in der Funktion MyStarter innerhalb von JavaCalls.c die Zeilen

sprintf(javaParameter[1], "-Xmx64m"); bzw.

sprintf(javaParameter[4], "-Xmx64m");

entsprechend angepasst werden. Anschließend ist das Projekt neu zu kompilieren.

## AIMSUN Scripting

Skripte ermöglichen die automatische Vorbereitung und Durchführung von Simulationsläufen. AIMSUN verfügt über einen Interpreter für Python. Damit dieser alle Standardbibliotheken von Python nutzen kann, ist jedoch eine zusätzliche Python-Installation erforderlich.

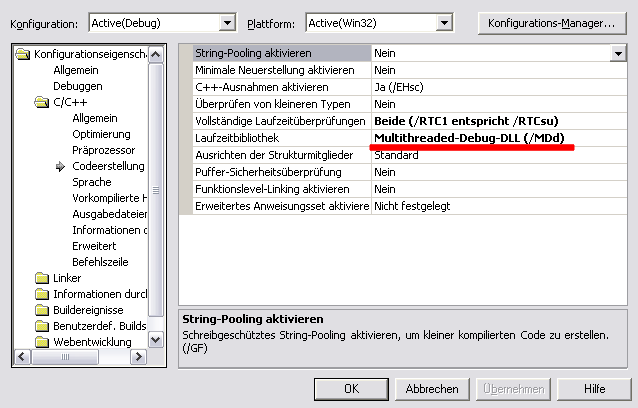
Es gibt die folgenden Möglichkeiten, Skripte unter AIMSUN einzusetzen:

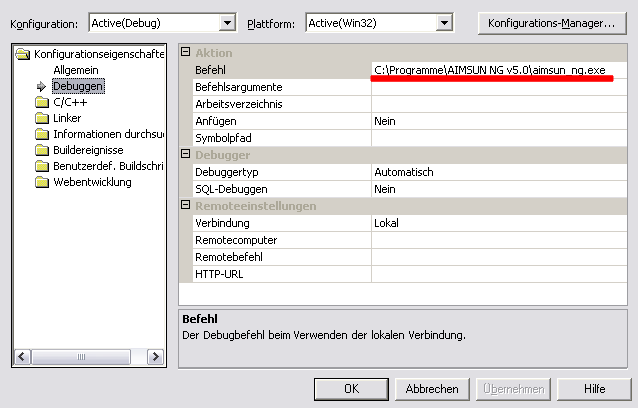
* AIMSUN kann aus einem Python-Skript heraus gestartet werden, wobei nach erfolgreichem Start über Skriptbefehle auf die verschiedenen Netzkomponenten zugegriffen werden kann. Diese Variante wird derzeit für die Ebene 2 des OTC-Systems verwendet. Der Start von AIMSUN aus dem Skript heraus ermöglicht den Einsatz eines Debuggers, Voraussetzung ist jedoch das Setzen einiger Umgebungsvariablen:
* Ein Skript kann beim Start von AIMSUN als Parameter an den Simulator übergeben werden. Das übergebene Skript wird dann unmittelbar nach dem Start durch den Simulator ausgeführt. Die notwendigen Befehle lauten *aimsun\_ng.exe –script <script.py>* bzw. *angconsole.exe –script <script.py>*, je nachdem, ob die „normale“ Benutzungsoberfläche oder die Konsolenversion von AIMSUN zu Einsatz kommen soll.
* Skripte können als Teil eines AIMSUN-Modells abgelegt und manuell in der graphischen Benutzungsoberfläche programmiert und ausgeführt werden, nachdem Simulator und Modell zuvor per Hand geladen wurden.

## Debuggen

### DLL, API („C-Seite“)

Es ist möglich, von Visual Studio aus, die erstellte API-DLL zu debuggen. Folgende Einstellungen sind dazu im Projekt vorzunehmen (sollte in der Konfiguration Debug schon passend eingestellt sein):

Unter C/C++ 🡪 Codeerstellung ist als Laufzeitbibliothek die Multithreaded-Debug-DLL anzugeben.

 Weiterhin ist unter Debuggen der aufzurufende Befehl anzugeben, hier also die EXE von AIMSUN.

In Visual Studio im Menü Debuggen auf Starten klicken (*F5*). Es startet AIMSUN. Hier ist manuell das passende Modell zu wählen und die Simulation zu starten, erst jetzt wird die DLL geladen, und der Debugger kann eingreifen.

Falls dies nicht klappt, ist die häufigste Ursache, dass vom Modell die falsche DLL (also eine DLL, die nicht gegen die Debug-DLL gelinkt wurde) geladen wird. Dann vorher die Debug-DLL erst neu erstellen und dann den Debug-Vorgang erneut starten.

### Debugging: Java

Da die JVM für den Controller und dessen GUI direkt aus der API-DLL heraus gestartet werden muss, ist für die Java-Seite nur Remote-Debugging möglich. Dazu muss die JVM mit den passenden Parametern gestartet werden. Für jede Replication ist einzeln einstellbar, ob eine Debugger-Schnittstelle geöffnet werden soll.

Dazu wurde ein neues Attribut *Attach Java Debugger* definiert (falls das Attribut oder der ganze Reiter *Attributes* in den Eigenschaften der Replications fehlen, muss für dieses Modell zunächst einmal eine Replication mit angehängter API-DLL initiiert werden, dann wird das Attribut erzeugt).

Setzt man das Attribut auf 0, wird keine Debug-Schnittstelle geöffnet, bei 1 wird die Schnittstelle geöffnet, bei 2 wartet die JVM zusätzlich nach dem Laden der Klassen, bis sich ein Debugger anhängt (nur so kann man zuverlässig auch Funktionen debuggen, die während der Initialisierung genutzt werden, z. B. zum Übertragen des Netzes).

Zu deren Nutzung öffnet man das zugehörige Java-Projekt, öffnet Run🡪Edit Configurations, und erstellt über New eine neue *Remote Configuration*. Sicherstellen, dass das richtige Projekt unter Project auftaucht, die übrigen Einstellungen sollten standardmäßig richtig gesetzt sein (Connection Type: Standard (Socket Attach), Host: localhost, Port: 8000).

Sobald jetzt die JVM gestartet ist, kann man sich über diese Launch Configuration mit dem Debugger an den laufenden Code anhängen, Breakpoints setzen, Variablen inspizieren, usw.

### Java: Debugging Layer 2

Öffnen Sie IntelliJ IDEA und ein Netzwerk in AIMSUN. Starten Sie die Simulation ohne Debug-Schnittstelle (Attribut = 0). Starten Sie Layer 2 nun mit der Debug-Batchdatei und hängen sich dann von IntelliJ mit dem in 1.4.2 erstellten Debug-Modus an.

### Java: Logging

Da mangels vorhandener Konsole einfache Ausgaben mittels System.out nicht möglich sind, wird im Projekt log4j verwendet. <http://logging.apache.org/log4j/docs/manual.html>. Konfiguriert wird über die Datei log4j.properties, die sich im Verzeichnis java\src befindet.

Im Code werden Log-Ausgaben wie folgt erzeugt:

**if** (*logger*.isDebugEnabled()) {

*logger*.debug("Logging-Text.");

}

Hier wird also die Ausgabe „Logging-Text“ an alle aktiven Appender ausgegeben, die mindestens das Level „Debug“ haben. Die if-Klammer kann auch wegfallen, sie bringt, falls nicht geloggt wird, ein wenig Performance. Je nach Level wäre dort entsprechend z. B. isWarnEnabled() zu verwenden.

### Debugging von Python-Skripten

Mit Hilfe von IntelliJ und des Plugins [PyDev](http://pydev.sourceforge.net/) ist ein Debugging des Skriptcodes möglich.

Um Python-Skripte debuggen zu können, müssen AIMSUNs Python-Bibliotheken eingebunden werden. Der hierfür notwendige Dialog kann über das Menü *Project – Properties – PyDev PYTHONPATH* erreicht werden. Mit Hilfe der Schaltfläche *Add source folder* kann der Ordner %AIMSUNFOLDER%\shared\scripts\libs\ in den Suchpfad aufgenommen werden.

Das Debugging erfolgt über den integrierten Debugger, der z.B. per Rechtsklick und Auswahl von Debug as🡪Python Run im Editorfenster des Skripts gestartet werden kann.

## Python, Java und Aimsun

Die Steuerung von AIMSUN für die Optimierungsläufe der Ebene 2 wird durch Python-Skripte erledigt. In diesem Fall sollte auch die JVM vom Skript gestartet werden. Daher liegt das Modul fertig kompiliert vor. Die Datei *javalauncher.dll* enthält ein Python-Modul zum Starten der JVM aus einem Skript heraus. Diese muss im Suchpfad von Python (sys.path) liegen. Für AIMSUN in %PATH\_TO\_AIMSUN%\shared\scripts\libs\*.* Sinnvoll wäre es, die Erweiterung von .dll auf .pyd zu ändern.

### Python-Modul erstellen

Die zu der von AIMSUN verwendeten Python-Version passenden Quellen in ein Verzeichnis entpacken (<http://www.python.org/download/>). Im Unterverzeichnis *PCbuild* befindet sich ein VC++-Workspace (pcbuild.dsw). Diesen mit Visual Studio öffnen, es folgt eine Konvertierung in das aktuelle Visual Studio Format. Das Teilprojekt *pythoncore* auswählen und sowohl als *Debug* wie auch als *Release* erstellen. Einige weitere Teilprojekte werden dabei ebenfalls erstellt, nicht alle erfolgreich, das ist kein Problem. Entscheidend ist, dass anschließend die Bibliotheken *pythonXX.lib* und *pythonXX\_d.lib* im Verzeichnis *PCbuild* liegen (XX = Versionsnummer).

Dann das Verzeichnis *PC\example\_nt* eine Ebene nach oben kopieren (auf die gleiche Ebene wie PC, Lib, …) und entsprechend dem eigenen Projekt umbenennen (oder das *javalauncher*-Projekt aus dem CVS verwenden, dann können die Anpassungen entfallen). Auch hier ist ein Workspace enthalten, der beim ersten Öffnen mit Visual Studio konvertiert wird. Im Projekt sollte überall „*example*“ durch einen zur eigenen Extension passenden Bezeichner ersetzt werden. Dieser ist auch von Hand in der Datei example.def anzupassen. Wenn diese Datei entsprechend dem Modulnamen umbenannt wird, müssen die Projekteigenschaften unter Linker/Eingabe/Moduldefinition angepasst werden. Weiterhin ist zu prüfen, dass unter Linker/Eingabe/Zusätzliche Abhängigkeiten auf die richtige *pythonXX.lib* bzw *pythonXX\_d.lib* (für Debug-Builds) verwiesen wird. Unter *Linker/Befehlszeile/zusätzliche Optionen* ist darauf zu achten, dass die richtige Funktion exportiert wird (gleiche Angabe wie in example.def, s. o.). Modul erstellen, ggf. umbenennen (auch in den Projekteigenschaften unter *Linker/Allgemein/Ausgabedatei* festlegbar), ins Verzeichnis *%*PATH\_TO\_AIMSUN*%\shared\scripts\* kopieren, und es sollte alles funktionieren.

# Starten des OTC-Systems

Die OTC-Architektur kann grob in drei Ebenen aufgeteilt werden:

* Ebene 0 umfasst den Verkehrsknoten mit der Lichtsignalanlage.
* Ebene 1 enthält eine Observer-Komponente, die die Verkehrslage am Knoten beobachtet und auswertet, sowie ein LCS, das für die Auswahl einer zur jeweils aktuellen Lage passenden Steuerung verantwortlich ist.
* Ebene 2 umfasst einen EA, der jeweils auf Anforderung Steuerungen für eine bestimmte Verkehrslage optimiert und diese der Ebene 1 zur Verfügung stellt.

Ebene 1 und 2 bilden separate Programmteile mit unterschiedlichen technischen Umsetzungen. Ebene 1 nutzt AIMSUNs API, um während einer laufenden Simulation Verkehrsdaten zu sammeln und Steuerungen anzupassen. Ebene 2 nutzt AIMSUNs Scripting-Möglichkeiten, um Steuerungen automatisiert bewerten zu können. Die Kommunikation zwischen den Ebenen findet mit Hilfe der „Remote Method Invocation“-Technik (RMI) von Java statt. Ebene 1 sendet Optimierungsaufträge an Ebene 2, die alle für die Optimierung wichtigen Daten (wie die z.B. die zu betrachtende Verkehrslage) enthalten. Ebene 2 übernimmt daraufhin die Optimierung und sendet das Optimierungsergebnis zurück an Ebene 1.

Im Folgenden ist kurz dokumentiert, welche Schritte zum Start von Ebene 1 und 2 notwendig sind.

## Ebene 1

Der Start der Ebene 1 erfolgt durch Laden des Netzmodells in AIMSUN, Hinzufügen der API-DLL (AAPI\_D.dll oder AAPI\_R.dll) zum Szenario (Site(s) 🡪 Scenarios 🡪 Scenario 🡪 Properties 🡪 AIMSUN-API) und Start einer interaktiven Simulation (Aimsun 🡪 Simulate).

Die API-DLL übernimmt das Starten einer JVM, in der die Ebene 1 ausgeführt wird. Nun sollte sich die Java-GUI des OTC-Managers öffnen. Von Ebene 1 wird eine RMI-Registry erzeugt, die den Kontakt zwischen Ebene 1 und Ebene 2 ermöglicht.

## Ebene 2

Nach Start von Ebene 1 kann Ebene 2 durch den Aufruf des Batch-Scripts im Verzeichnis Aimsun\_Modell

opt\_InternalFTC8.bat <x> <ANG-Filename>

gestartet werden, wobei <x> die Anzahl der zu startenden EAs angibt (Stand 06/2016: wegen der aktuellen Lizenz geht nur das Ausführen einer Instanz). Die Batchdatei startet eine JVM, in der Ebene 2 ausgeführt wird. Ebene 2 versucht zunächst, sich an einer auf dem lokalen Rechner laufenden RMI-Registry anzumelden. Falls dies scheitert, wird der Benutzer aufgefordert, den Rechner anzugeben, auf dem die Ebene 1 – und damit die Registry – läuft.

## Ebene 1 und Ebene 2 auf verschiedenen Rechnern

Das OTC-System kann auf zwei Rechner verteilt werden: Auf einer Maschine arbeitet die Ebene 1, auf der zweiten die Ebene 2.

Ist auf einem der beteiligten Rechner kein Visual Studio installiert, so ist zu beachten, dass dort nicht die Debug-, sondern die Release-Version der API-DLL und der JavaNativeEvent.dll verwendet werden, da anderenfalls einige in Visual Studio enthaltene und dynamisch eingebundene Bibliotheken fehlen.