Universität Koblenz-Landau
Abteilung Koblenz

Institut für Wirtschafts- und Verwaltungsinformatik
Prof. Dr. Klaus G. Troitzsch

Studienarbeit

EnSiCC

(Environment for Simulation of Communication Centers)

Bodo Hinterwäller
dachs@uni-koblenz.de

Ulf Lotzmann
ulf@uni-koblenz.de

2. Fassung

Koblenz, 25 April 2002

Inhaltsverzeichnis

1. Kontext der Studienarbeit

1.1 CCIRP (Communication Center Initiative Rheinland-Pfalz)

CCIRP ist ein Projekt des Landes Rheinland-Pfalz in Zusammenarbeit mit der Universität Koblenz und beschäftigt sich mit der Durchführung einer Landesinitiative im Bereich Communication Center mit dem Ziel der Verfolgung von Industrieansiedlungsstrategien, der Schaffung neuer Arbeitsplätze und der Stärkung des Standortes Rheinland-Pfalz (siehe *www.uni-koblenz.de/~ccirp*). Das Projekt CCIRP wird gefördert durch die Ministerien für Arbeit, Soziales und Gesundheit sowie für Wirtschaft, Verkehr, Landwirtschaft und Weinbau.

Ein Teilprojekt von CCIRP ist der Aufbau eines Communication Center Referenzlabors zur Gewinnung vertiefter Erkenntnisse, dessen Aufgabe unter anderem die Simulation von Communication Center Prozessen ist. In diesem Kontext wurde die hier vorliegende Studienarbeit unter Leitung von Prof. Dr. Klaus G. Troitzsch entwickelt.

1.2 Vorprojekt

Grundlage für die vorliegende Studienarbeit ist die im Januar 2000 fertiggestellte Diplomarbeit von Gregor Niehl. Diese wurde im Rahmen des CCIRP-Vorprojekts entwickelt und beschäftigt sich mit der Erstellung eines Prototypen zur Simulation eines Call Centers.

Zudem dient sie als Grundlage für das Verstehen und Diskutieren der Geschäftsprozesse eines Call Centers, welche unter Anwendung von EPKs modelliert wurden. Die Simulation umfasst die Implementierung der aufgestellten Modelle mittels Simple++. Dies ist eine Software zur graphischen und integrierten Modellerstellung, die zur Unterstützung von Simulation und Visualisierung von Produktionssystemen und –prozessen verwendet wird.

2. EnSiCC (Environment for Simulation of Communication Centers)

2.1 Aufgaben und Ziele

Die Weiterentwicklung des bereits vorliegenden Prototyps von Gregor Niehl ist ein primäres Ziel dieser Studienarbeit. Da Simple++ und die darin enthaltene Sprache Simpletalk nur bedingt für die Simulation von Call Centern geeignet ist und zudem als kommerzielles Produkt vorliegt, wurde die Entscheidung getroffen, von dieser Modellerstellungs-Umgebung Abstand zu nehmen und ein neues Simulationswerkzeug unter Zuhilfenahme der Programmiersprache Java zu entwickeln. Dabei ergaben sich für die Erstellung der Studienarbeit zwei zyklische, ineinander übergehende Phasen: einerseits das Entwerfen und Implementieren eines neuen Simulationswerkzeugs, andererseits das Übertragen der Geschäftsprozessmodelle auf die neue Simulationsumgebung sowie deren Evaluation. Daraus entstand das in dieser Studienarbeit entwickelte Simulationssystem EnSiCC (Environment for Simulation of Communication Centers), welches in den folgenden Abschnitten näher vorgestellt wird.

2.2 Aufbau der Simulationsumgebung

EnSiCC stellt eine graphische Oberfläche (aufbauend auf Klasse SL_Window) zur Verfügung, mit der der Benutzer Simulationsmodelle erstellen kann. Zur übersichtlichen Darstellung der Objekthierarchie und als Hilfsmittel zur Navigation und Konfiguration steht ein Objektbrowser (Klasse SL_Browser) zur Verfügung (siehe Abbildung 1). Zudem existieren kontextbezogene Dialoge zur Konfiguration der einzelnen Komponenten.

Die zu simulierenden Prozesse werden durch Warteschlangen-Stationen-Netzwerke (Klassen SL_Network und SL_NetFrame) repräsentiert (siehe Abbildung 1). Diese bestehen aus Bearbeitungsstationen, die durch Konnektoren verknüpft sind.

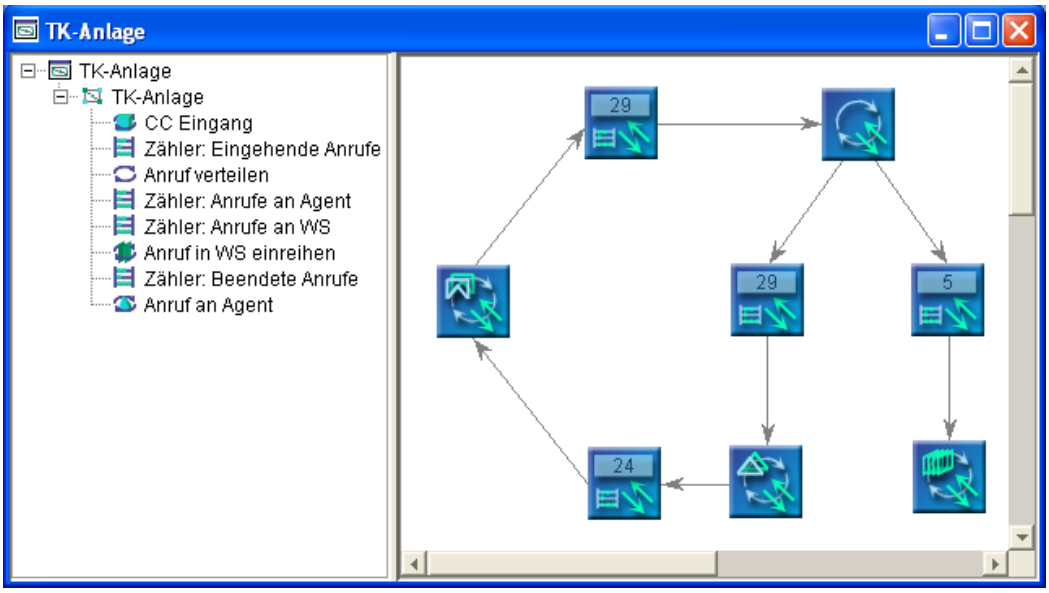


Abbildung 1 Fenster mit Objektbrowser und Netzwerk

Stationen (Klasse SL_Station) erzeugen und bearbeiten bewegliche Datenobjekte (sogenannte Moveable Entities, kurz: MEs; Klasse SL_MoveableEntity), wobei die durchzuführenden Aktionen mittels benutzerdefiniertem Java-Code festgelegt werden. Diesen MEs lassen sich durch die Stationen beliebige Informationen hinzufügen und zu späteren Zeitpunkten wieder entnehmen. Mittels Konnektoren (Klasse SL_Connector), die gleichzeitig die Funktionalität von Warteschlangen aufweisen, können die MEs an nachfolgende Bearbeitungsstationen weitergereicht und bei Bedarf durch Animation visualisiert werden.

Das Simulationsmodell eines größeren Systems besteht in der Regel aus mehreren Teilnetzwerken, wobei die einzelnen Netzwerke abgeschlossene Einheiten zur Simulation von Teilprozessen darstellen. Um Daten zwischen diesen Teilnetzen austauschen zu können, wurde das „Provider-Consumer-Konzept“ entwickelt. Dieses beinhaltet ausgezeichnete Stationen, die konnektorlose Kommunikationsverbindungen bereitstellen (Klassen SL_Provider und SL_Consumer).

Die zentrale Komponente zur Steuerung eines Simulationsablaufs ist der Event Controller (Klasse SL_EventController, siehe Abbildung 2). Hierüber hat der Benutzer einerseits Zugriff auf die Ablaufgeschwindigkeit der Simulation und kann die Animation beeinflussen, andererseits kann er Informationen über die laufende Simulation erhalten. So können beispielsweise aktuelle Zählerstände (Klasse SL_Counter) und Werte von Variablen (Klasse SL_Variable) eingesehen sowie vorher zeitlich eingestellte Ereignisse überwacht werden.

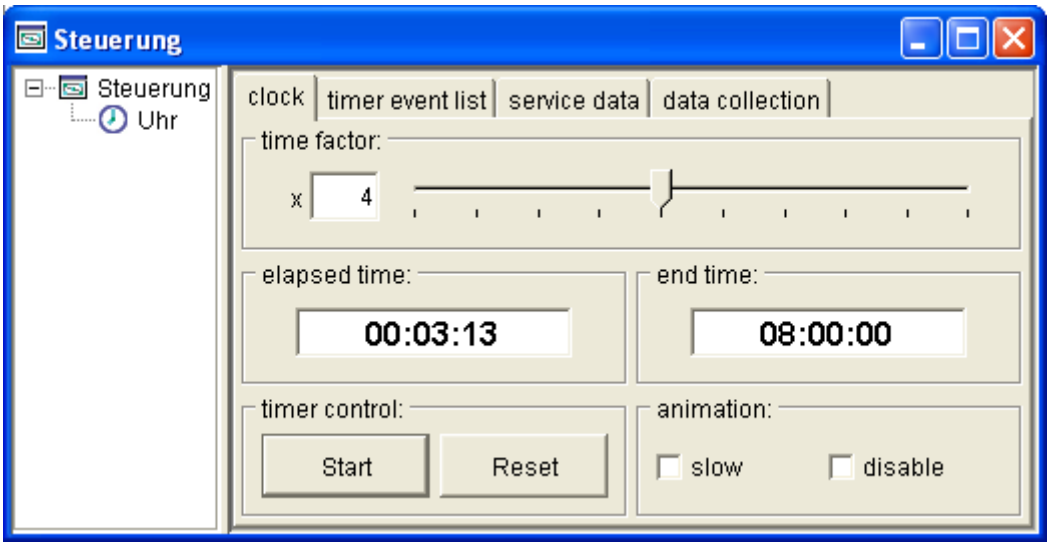


Abbildung 2 Event Controller

Wie bereits erwähnt, bestehen die einzelnen Bereiche eines Communication Center Modells innerhalb der EnSiCC-Simulationsumgebung aus Netzwerken. Eine Erweiterung dieser Standard-Netzwerke ist der Multi-Netframe (Klasse SL_MultiNetFrame, siehe Abbildung 3). Damit können beispielsweise die durchzuführenden Prozesse eines Agenten einmalig vom Benutzer implementiert und zur Laufzeit beliebig viele Kopien dieses Netzwerks erstellt werden, um eine Vielzahl von Communication Center Agenten zu simulieren.

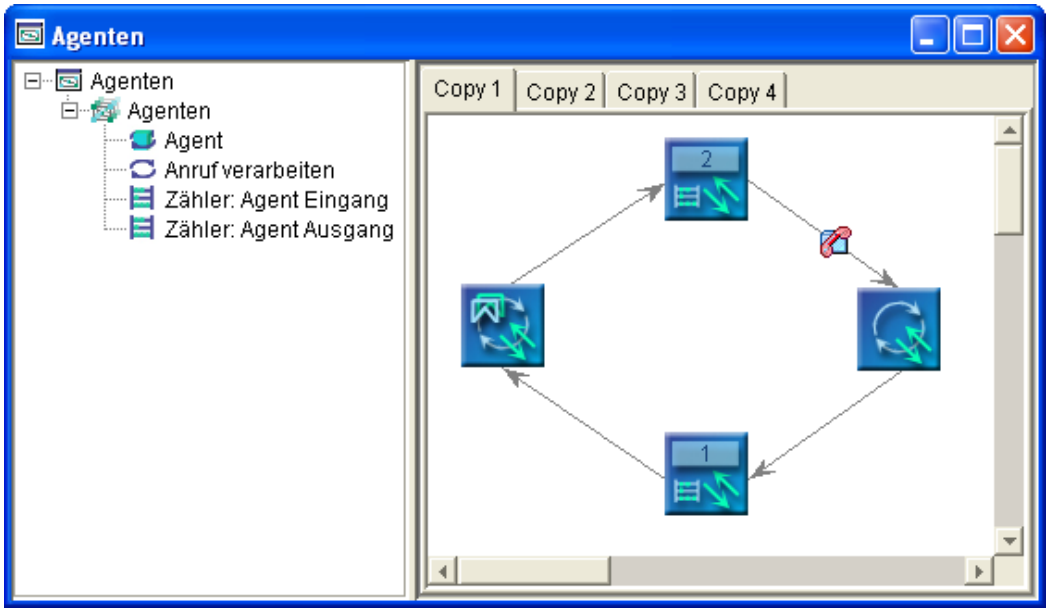


Abbildung 3 Multi-Netframe

Um auf externe Daten für eine Simulation zugreifen zu können, besteht die Möglichkeit einer Datenbankanbindung durch eine Datenstation (Klasse SL_DataStation) und ein Datenmodul (Klasse SL_Table). Die Datenbank wird in einem separaten Fensterbereich graphisch dargestellt (siehe Abbildung 4), wobei die während des Simulationslaufs benutzten Datenelemente als gesperrte Objekte gekennzeichnet werden. Mit dieser Art der Dateneinbindung lässt sich eine Simulation beispielsweise mit realen Kundendaten durchführen.

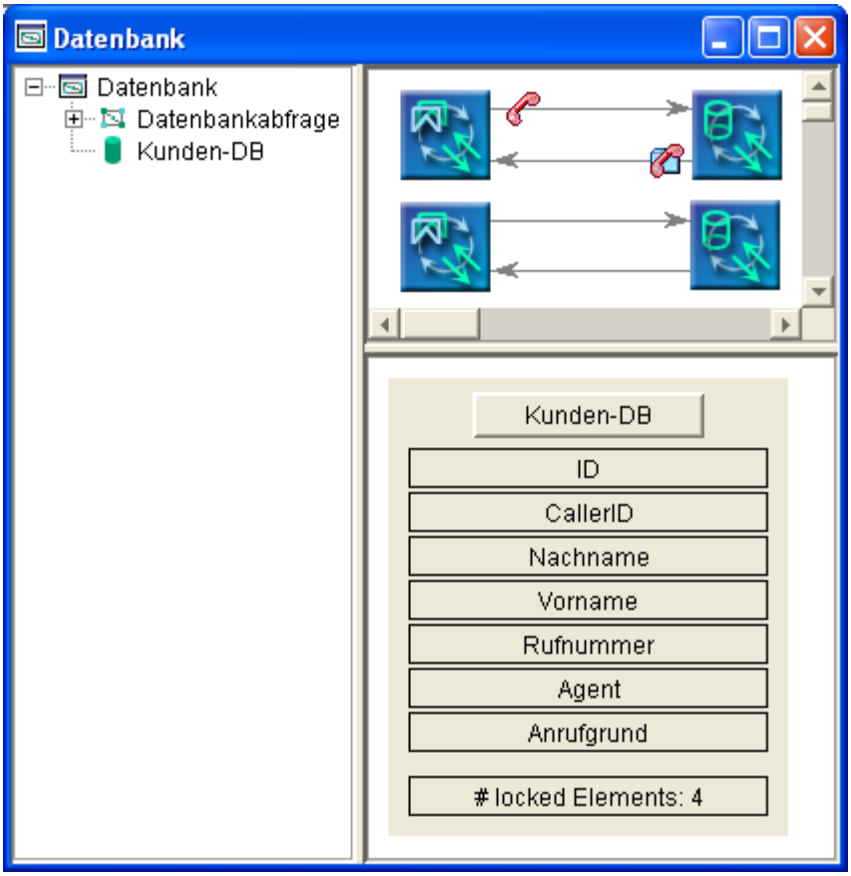


Abbildung 4 Datenmodul (rechts unten) und Netzwerk zur Datenbankabfrage (rechts oben)

Zur statistischen Auswertung der Simulations(-zwischen)-Ergebnisse stehen die Diagramm-Typen Plotter (Klasse SL_Plotter) und Chart (Klasse SL_Chart) zur Verfügung (siehe Abbildung 5). Diese Komponenten greifen auf die vom Benutzer in die Netzwerke eingebauten Zähler (Klasse SL_Counter) und Variablen (Klasse SL_Variable) zu und vermitteln schon während der Laufzeit der Simulation einen Überblick über die zu sammelnden Daten, wobei jederzeit Datenquellen hinzugefügt oder entfernt werden können. Zudem ist es möglich, anhand mathematischer Ausdrücke mehrere Variablen und/oder Zähler

miteinander in Beziehung zu setzen. Nach Ablauf steht dem Benutzer eine Statistik über den durchgeführten Simulationslauf zur Verfügung.

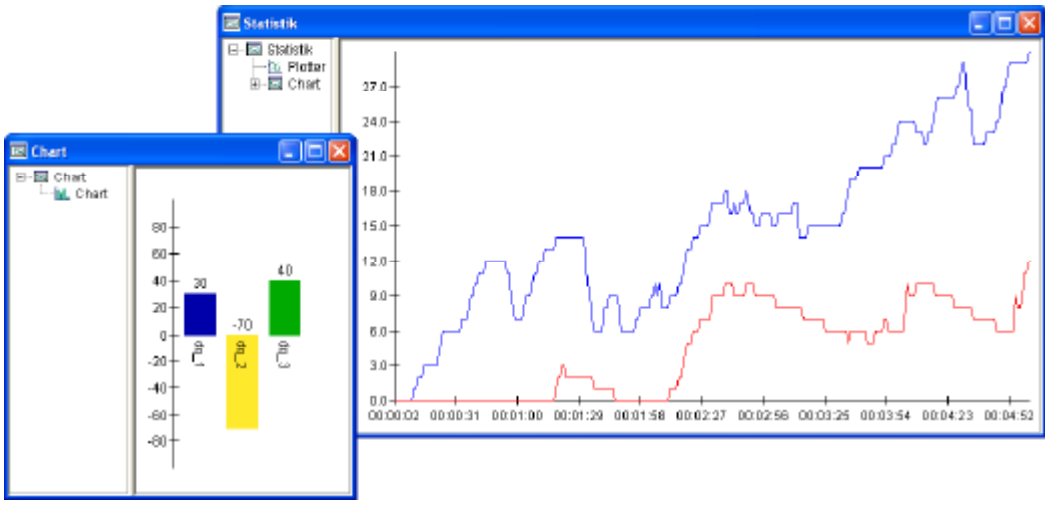


Abbildung 5 Diagrammtypen: Chart und Plotter

2.3 Implementation

Abbildung 6 zeigt das Klassendiagramm der erstellten EnSiCC-Simulationsumgebung.

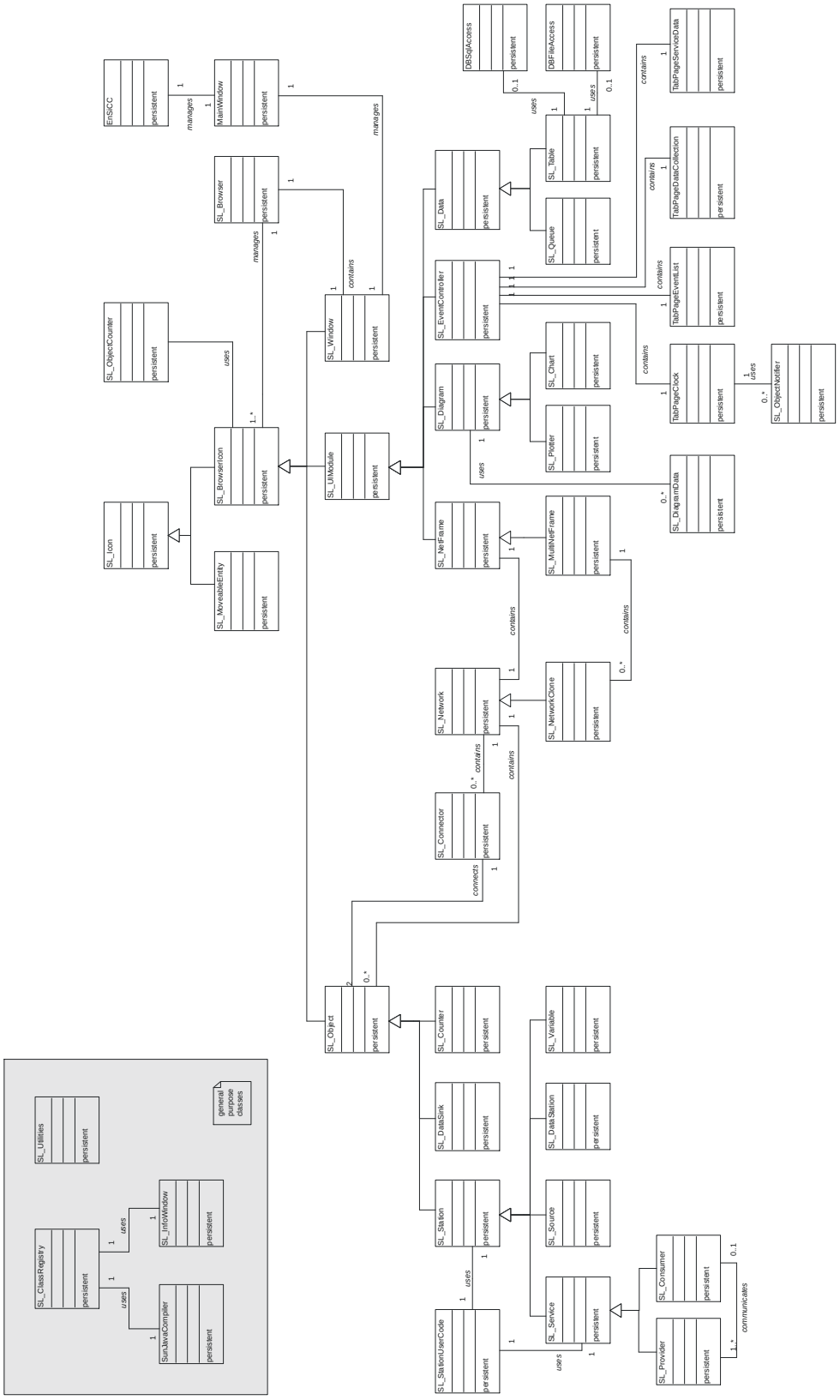


Abbildung 6 Klassendiagramm EnSiCC (Ausschnitt)

2.3.1 SL_Icon

Die Klasse SL_Icon ist die Wurzelklasse der entwickelten Simulationsbibliothek. Sie repräsentiert das Haupticon (mainIcon) des GUI-Objekts und verwaltet dessen Namen. SL_Icon stellt grundlegende Methoden zur Verfügung, die von abgeleiteten Klassen gegebenenfalls überschrieben werden müssen: Durch updateGUIObj() wird das Aktualisieren der graphischen Darstellung veranlasst, mittels addContextMenuItems() können objektspezifische Einträge in ein Kontextmenü eingefügt werden, in register()/unregister() werden Aufrufe zur Registrierung bzw. Entfernen der Registrierung (siehe SL_EventController) zusammengefasst und destroy() implementiert die Aktionen zum Entfernen eines GUI-Objekts. Der (Haupt-) Konstruktor von SL_Icon erwartet als Parameter Objekt-Name und Name der Bilddatei des darzustellenden Icons. Weiterhin existiert ein Konstruktor, welcher über einen Parameter-String das Objekt konfiguriert und zum Einladen von gespeicherten Simulationsbeschreibungen benötigt wird (siehe SL_ClassRegistry). Dieser Konstruktor muss von allen abgeleiteten Klassen überschrieben werden.

2.3.2 SL_BrowserIcon

Die Klasse SL_BrowserIcon erweitert SL_Icon um die Fähigkeit zum Einfügen des GUI-Objekts in einen Objekt-Browser. Zu diesem Zweck stellt SL_BrowserIcon Funktionalitäten zur Verfügung, die Objekten dieser Klasse verschiedene Icons für die graphische Darstellung im Objekt-Browser (siehe SL_Browser) zuweisen. Weiterhin existieren Methoden zum Markieren (mark() bzw. unmark()) des GUI-Objekt-Icons durch Benutzer-Mausklick oder Aktivierung im Objektbrowser. Zudem verwaltet SL_BrowserIcon eine Objekt-ID, die der eindeutigen Identifizierung aller GUI-Objekte dient und von SL_ObjectCounter vergeben wird. Die Konstruktoren erwarten als Parameter Objekt-Name, gegebenenfalls Name der Bilddatei des darzustellenden Icons sowie den Elternknoten für die Darstellung im Objekt-Browser.

2.3.3 SL_Object

SL_Object bildet die Elternklasse aller Objekte in Simulationsnetzwerken. Sie enthält Methoden zum Hinzufügen (addInputConnector(), addOutputConnector()) und Entfernen

(removeInputConnector(), removeOutputConnector()) von ein- und ausgehenden Konnektoren, zum Empfangen (notyfyFetchME(), fetchME()) und Versenden (sendME()) von MEs und implementiert den ME-Verarbeitungsthread (Methoden run() und processME()).

Der ME-Verarbeitungsprozess läuft in folgenden Schritten ab:
1. SL_Object versucht, von einem der einlaufenden Konnektoren ein ME zu holen (fetchME()). Falls kein ME vorhanden ist, geht SL_Object für einen definierten Zeitraum in einen 'wait'-Zustand. Dieser Zustand kann vorzeitig verlassen werden, wenn ein einlaufender Konnektor (mittels notifyFetchME()) das Vorhandensein eines MEs signalisiert.
2. Das empfangene ME wird zur Verarbeitung an processME() übergeben. Die Verarbeitungsdauer von processME() wird ermittelt.
3. Um eine definierte gleichmäßige Gesamtverarbeitungsdauer zu gewährleisten, wird der Thread für die Zeitdifferenz zur festgelegten Verarbeitungsdauer (duration) angehalten (mittels sleep()-Methode).

2.3.4 SL_Station

SL_Station bildet die Elternklasse aller konfigurierbaren Verarbeitungsstationen in Simulationsnetzwerken. Im Mittelpunkt steht die Verwaltung des Benutzer-Sourcecodes sowie des Benutzercode-Objekts. Der Benutzercode setzt sich aus userCode_processME, userDeclarations und userInitializations zusammen und kann über Konfigurationsdialoge (DlgPagePropertyEditor und DlgPageUserCodeEditor) durch den Benutzer angepasst werden. internalDeclarations und internalInitializations werden zur Anpassung erweiterter, von SL_Station abgeleiteter Stationen herangezogen. Das Kompilieren des Benutzer-Sourcecodes wird mittels SL_ClassRegistry durchgeführt. Zum Laden des kompilierten Benutzercodes stellt SL_Station die Methode loadUserCode() zur Verfügung. Die Methode processME() leitet das übergebene ME zur Bearbeitung an das Benutzercode-Objekt (userCodeObject) weiter.

2.3.5 SL_Source

Die Klasse SL_Source implementiert die Quelle für bewegliche Datenobjekte (MEs). Sie erzeugt in regelmäßigen, durch den Benutzer definierbaren Abständen SL_MoveableEntity-Objekte, die durch den Benutzercode des SL_Source-Objekts bearbeitet werden. Das nach Eintreffen eines 'timer events' vom EventController erzeugte ME wird in eine interne Warteschlange (queue) eingereiht. Anschließend wird durch Aufruf von notifyFetchME() der Verarbeitungsprozess angestoßen. Die interne Warteschlange ist notwendig, um zu verhindern, dass bei einem "Stau" auf den ausgehenden Konnektoren das gesamte System blockiert wird. SL_Source überschreibt die SL_Station-Methode fetchME(), sodass das zu bearbeitende ME aus queue (statt eingehendem Konnektor) entnommen wird. SL_Source akzeptiert im Gegensatz zu anderen Bearbeitungsstationen keine eingehenden Konnektoren.

2.3.6 SL_DataStation

Die Klasse SL_DataStation ermöglicht den Zugriff auf Objekte vom Typ SL_Queue oder SL_Table und somit auf die von diesen Klassen zur Verfügung gestellten Datenzugriffsmöglichkeiten. Nachdem dieser Station eine 'Quell-Station' (Table oder Queue) zugewiesen wurde, ändert sich die Icon-Darstellung des GUI-Objekts.

2.3.7 SL_Variable

Die Klasse SL_Variable erweitert SL_Station um eine ausgezeichnete Double-Variable (variable). Diese kann durch den Benutzercode gesetzt (setVariable()) und ausgelesen (getVariable()) werden. SL_Variable implementiert das Interface SL_DataSourceInterface und kann damit als Datenquelle für Diagramme dienen (siehe SL_Counter).

2.3.8 SL_Service

Die Klasse SL_Service ist die Vaterklasse von SL_Consumer und SL_Provider und implementiert die Funktionalität, gesendete MEs empfangen zu können (receiveME()). Durch die Klasse DialogStationConfig können die abgeleiteten Objekte SL_Consumer und SL_Provider konfiguriert sowie der Usercode eingegeben werden.

2.3.9 SL_Provider

Die Klasse SL_Provider nimmt MEs von SL_Consumer-Objekten entgegen und gibt diese über den Standardausgang an das angrenzende Netzwerk weiter. Die SL_Provider-Objekte werden in TabPageServiceData registriert (siehe register()-Methode), um den Zugriff seitens der Consumer zu gewährleisten. Das Annehmen von MEs (receiveME()) wird von der Vaterklasse SL_Service geerbt. Damit keine weiteren MEs angenommen werden können, wird das Provider-Objekt gesperrt (lockProvider()). Sobald die ME-Bearbeitung im angrenzenden Netzwerk vollendet ist und das ME wieder bei dem Provider-Objekt angekommen ist, wird es an den Quell-Consumer zurückgegeben und das Provider-Objekt entsperrt (unlockProvider()).

2.3.10 SL_Consumer

Die Klasse SL_Consumer dient zum Senden von MEs an Objekte vom Typ SL_Provider, wobei die beiden beteiligten Station nicht durch Konnektoren miteinander verbunden sein müssen. Dadurch ist es möglich, MEs aus einem Netzwerk-Fenster in ein anderes zu transferieren. Zu diesem Zweck wird die Methode transmitME() mit dem zu übergebenden ME sowie dem Ziel-Provider als Parameter aufgerufen.

2.3.11 SL_Counter

Die Klasse SL_Counter implementiert das Interface SL_DataSourceInterface und kann damit als Datenquelle für Diagramme dienen (siehe auch SL_Variable). SL_Counter erzeugt eine Instanz einer Zähl-Station, welche die einlaufenden MEs zählt und zum Standardausgang weiterleitet. Für die statistische Auswertung eines Simulationslaufs

werden diese Counter zentral registriert (siehe Methode register()), sodass die Diagrammklassen SL_Chart und SL_Plotter auf diese Datenquellen zugreifen können. Weiterhin wird der aktuelle Zählerstand in der graphischen Darstellung der Counter während eines Simulationslaufs ständig angezeigt und aktualisiert. Über das Kontextmenü hat der Benutzer zudem die Möglichkeit, einen Counter-Reset (Zurücksetzen des Zählers auf Null) durchzuführen (siehe Methode resetCounter()).

2.3.12 SL_DataSink

Die Klasse SL_DataSink implementiert eine 'Datensenke'. Dieses ausgezeichnete Netzwerk-Objekt wird zum Entfernen von nicht mehr benutzten MEs herangezogen.

2.3.13 SL_Connector

Die Klasse SL_Connector implementiert einen Konnektor, der als (graphischer) Verbinder zweier Stationen sowie Transport- und Zwischenspeichermedium (Warteschlange) für MEs in Simulations-Netzwerken (SL_Network) dient. Der Transport der MEs wird bei Bedarf durch Animation visualisiert. Die Kapazität der Warteschlange und das Netzwerk, zu dem der Konnektor gehören soll, werden beim Erzeugen des Konnektor-Objekts (mittels getInstance()) als Parameter übergeben. Das Positionieren des Konnektors, das interaktiv durch den Benutzer erfolgt, wird nach Erzeugen des Konnektor-Objekts erstmals initiiert und kann zu beliebigen Zeitpunkten später erneut durchgeführt werden. Das "An- und Abkoppeln" des Konnektoranfangs bzw. -endes an eine bzw. von einer Station wird automatisch erkannt und löst entsprechende Aktionen bei Konnektor (Methoden onNetworkMouseMoved(), onNetworkMouseReleased()) und Station aus. SL_Connector besitzt die Methode putME(), mit der die Ursprungs-Station (Station am Konnektoranfang) dem Konnektor ein ME übergibt. Diese Methode reiht das ME in den 'Vector' incomingMEs ein und aktiviert anschließend den Animations-Thread, der die Durchführung der Transport-Animation steuert (dieser Schritt entfällt bei abgeschalteter Animation). Nach Beendigung der Animation erfolgt das Einreihen der ME in die Warteschlange (queue). Die maximale Anzahl der zu einem Zeitpunkt im Konnektor gespeicherten MEs wird durch queueSize begrenzt. Bei Erreichen der Kapazitätsgrenze geht der Konnektor in den Zustand "Gesperrt" über (Abfrage durch Methode isLocked()). Sobald ein ME in der leeren Warteschlange abgelegt wurde,

benachrichtigt der Konnektor die Ziel-Station (Station am Konnektorende), indem er deren Methode notifyFetchME() aufruft. Wenn die Ziel-Station die evtl. gerade laufende Bearbeitung abgeschlossen hat, wird sie versuchen, ein ME dem Konnektor zu entnehmen. Sie bedient sich dabei der Methode getME(). Nach Entnahme eines MEs wird ggf. die Sperrung des Konnektors wieder aufgehoben.
Neben der Schnittstelle zu Ursprungs- und Ziel-Station besteht eine enge Verbindung zum zugehörigen SL_Network-Objekt, da der Konnektor die (Maus-) Ereignisse sowie Anforderungen zum Starten oder Beenden des Editier-Modus von diesem bezieht.

2.3.14 SL_UIModule

Die Klasse SL_UIModule ist der Ursprung aller GUI-Komponenten, die in eine SL_Window-Gitterstruktur eingefügt werden können. Die Methode getInsertion() liefert die einzufügende Komponente, die in der Regel ein scrollPane-Objekt darstellt. Soll für abgeleitete Klassen eine andere Komponente eingefügt werden, muss die Methode getInsertion() entsprechend überschrieben werden. Um die Markierung eines SL_UIModule zu visualisieren, verwaltet die Klasse zwei verschiedene Border-Objekte, die je nach Zustand aktiviert werden. Der Konstruktor erwartet neben den in SL_BrowserIcon definierten Parametern die Einfüge-Position in die virtuelle Gitterstruktur des darstellenden SL_Window-Objekts.

2.3.15 SL_NetFrame

Die Klasse SL_NetFrame implementiert ein in SL_Window integrierbares UI-Modul, das ein SL_Network kapselt. Die für die Steuerung des SL_Network-Objekts notwendigen Aktionen (Erzeugen von Stationen, Konnektoren usw.) werden in SL_NetFrame (durch addContextMenuItems()) dem Kontextmenü hinzugefügt.

2.3.16 SL_MultiNetFrame

Die Klasse SL_MultiNetFrame implementiert ein in SL-Window integrierbares UI-Modul, das ein Master-SL_Network-Objekt sowie beliebig viele Kopien davon (SL_NetworkClone) kapselt.

Die Anzahl der Kopien variiert während eines Simulationslaufs nach einem durch den Benutzer festgelegten Schema. Die Konfiguration erfolgt über eine spezielle Dialog-Seite (DlgPageMultiNetFrameConfig). Die in den Netzwerk-Kopien enthaltenen Simulations-Objekte erhalten beim Erzeugen eindeutige, vom Master-Netzwerk abgeleitete Titel und Objekt IDs. Die Darstellung der Netzwerk-Kopien erfolgt in einer JTabbedPane-Komponente.

2.3.17 SL_Network

Die Klasse SL_Network dient der Darstellung und Verwaltung von Simulations-Netzwerken. Sie unterstützt das Editieren des verwalteten Netzwerks, indem sie Methoden zum Hinzufügen (addObject(), addConnector()) und Entfernen (removeObject(), removeConnector()) von Simulations-Objekten und Konnektoren, zum Editieren der Konnektoren (connectorEditComplete(), resetConnectorEdit()) sowie weitere Hilfsmethoden (getHitObject()) zur Verfügung stellt. Die Editierbarkeit des gesamten Netzwerks durch den Benutzer kann über setEditable() gesteuert werden.

2.3.18 SL_Data

Die Klasse SL_Data beinhaltet die Methode registerDataModule(), welche Objekte vom Typ SL_Table sowie SL_Queue in einem Vektor registriert, um SL_DataModule-Objekten Zugriff auf diese Objekte zu ermöglichen.

2.3.19 SL_Queue

Die Klasse SL_Queue implementiert eine Warteschlange, in welche MEs durch SL_DataStation-Objekte eingereiht (enqueue()) und entnommen (dequeue()) werden können. Dabei übernimmt SL_Queue die Verwaltung der Warteschlange und die Klasse QueuePanel die graphische Darstellung der MEs in der Warteschlange. Die aktuelle Anzahl enthaltener ME-Objekte wird in der graphischen Darstellung der Queue angezeigt und bei Bedarf aktualisiert (displayQueueSize()).

2.3.20 SL_Table

Die Klasse SL_Table ermöglicht den Zugriff auf Kundendatensätze, die durch eine Datenbank der Simulationsumgebung zugänglich gemacht werden. Einerseits kann auf eine Access-Datenbank mittels JDBC-ODBC und auszuführenden SQL-Befehlen zugegriffen werden (siehe DBSqlAccess). Da diese Variante aufgrund schlechter Performance (Zugriffszeiten) für die Simulation nur bedingt einsetzbar ist, wurde andererseits die Klasse DBFileAccess implementiert, welche die Daten in einem Array zur Verfügung stellt und zuvor aus einer Datei lädt. Das Zuweisen dieser Datenquellen geschieht mittels eines Konfigurationsdialogs, welcher über das Kontextmenü dieser Klasse aufgerufen wird (siehe DlgPageDBAccess). Damit der Benutzer nicht auf die in der Datenquelle angegebenen Spaltennamen angewiesen ist, kann er über den Konfigurationsdialog eigene Spaltenbezeichnungen angeben und auf diese während der Simulation zugreifen. Der Zugriff auf eine zugewiesene Datenquelle geschieht durch Datenstationen (siehe SL_DataStation), wobei das Umsetzen der benutzerdefinierten Spaltennamen auf die zugrundeliegende Datenquelle von der Methode mapColumnName() übernommen wird. Weiterhin stehen Methoden zum Selektieren (selectData()) und Einfügen (insertData()) von Datensätzen zur Verfügung. Zudem wird der Zugriff auf Datensätze, welche gerade von der Simulationsumgebung bearbeitet werden, gesperrt (lockElement()) und erst nach einer erfolgreichen Aktualisierung wieder freigegeben (unlockData()). Die graphische Darstellung dieser Klasse enthält die zugewiesenen Spaltennamen der Datenquelle sowie die Anzahl der zur Zeit gesperrten (d.h. in Bearbeitung befindlichen) Datensätze. Zudem implementiert SL_Table das Interface SL_DBInterface, welches die benötigten Methoden für den Datenzugriff auf die Klassen DBSqlAccess und DBFileAccess enthält.

Die Klasse DBSqlAccess ermöglicht den Zugriff auf für die durchzuführende Simulation benötigten Daten, die in Form einer Datenbank dem System zugänglich gemacht werden. DBSqlAccess stellt Methoden zum Zugriff auf Access- Datenbanken zur Verfügung, wobei eine ODBC-JDBC-Brücke erzeugt wird. Die Klasse SL_Table baut auf diese SQL-Datenbankanbindung auf und hält für die Simulationsumgebung die benötigten Daten bereit. Da der Datenbankzugriff bei erhöhter Simulationsgeschwindigkeit jedoch zu Performance-Problemen führt, wurde die Klasse DBFileAccess entwickelt, welche einen Datenbankzugriff lediglich simuliert und dabei auf Daten aus einem Array bzw. Daten aus einer Textdatei zurückgreift.

Die Klasse DBFileAccess simuliert einen Datenbankzugriff, indem sie Datensätze in einem Array bereitstellt, welches (wenn nötig) zur Laufzeit um einen vorher festgelegten Wert (upsizeValue) vergrößert wird. Weiterhin können Datensätze ausgelesen (select()), eingefügt (insert()) sowie verändert (update()) werden. Die Datensätze werden bei der Initialisierung eines Objekts dieser Klasse aus einer Datei (siehe dataBaseFileName und loadDataBaseFromFile()) gelesen, woraus das zugrundeliegende Array erzeugt wird. Nach Beenden der Simulation werden die geänderten und neu erstellten Datensätze in diese Datei zurückgeschrieben.

2.3.21 SL_Diagram

Die Klasse SL_Diagram stellt Daten innerhalb eines vom Benutzer zu wählenden Diagrammtyps (siehe SL_Plotter und SL_Chart) graphisch dar. Datenquellen können über einen Benutzerdialog dem Diagramm zugänglich gemacht sowie modifiziert werden. Weiterhin können das Aktualisierungsintervall sowie die darzustellende Diagrammfarbe über einen Benutzerdialog (siehe DialogStationConfig) zugewiesen werden.

SL_DiagramData bildet die Elternklasse für durch Benutzercode angepasste Datenquellen-Objekte, die bei SL_Diagramm-Objekten Verwendung finden. Der Java-Sourcecode der durch userCode angepassten, von SL_DiagramData abgeleiteten Benutzercode-Klasse wird durch die folgenden Klassenmethoden erzeugt: generateDiagramUserCode() generiert Code für eine "einfache" Diagramm-Datenquelle, während generateBufferedDiagramUserCode() Code für eine "gepufferte" Datenquelle zurückliefert. Die Besonderheit dieses Datenquellen-Typs ist, dass bei jedem Aufruf der Methode getValue() ein Wertepaar (übergebener Parameter, Ergebnis) im Vektor history abgelegt wird (im Benutzercode ist darauf zu achten, dass der Variablen result das Berechnungsergebnis zugewiesen wird). Die Abfrage der history ermöglicht die Methode values(). Einem SL_DiagramData-Objekt kann ein Datentyp zugewiesen (setDataType()) und (durch getDataType()) abgefragt werden, der ggf. Auswirkung auf die Skalenbeschriftung des zugehörigen Diagramms hat. Folgende Datentypen werden unterschieden: DATATYPE_DEFAULT, DATATYPE_INT, DATATYPE_DOUBLE und DATATYPE_TIME. Den Status des Datenquellen-Objekts ermittelt man über getStatus(), wobei STAT_VALID, STAT_INVALID und STAT_OBSOLETE als Ergebnis auftreten können.

2.3.22 SL_Chart

Die Klasse SL_Chart ermöglicht die Darstellung von Balkendiagrammen für eine statistische Auswertung eines Simulationslaufs. Dazu bedient sie sich der Klasse BarGraph zur Darstellung der einzelnen Balkenelemente.

Die Klasse BarGraph erzeugt Balkenelemente entsprechend den in einem Vektor (siehe dataVector) vorliegenden Datenquellen sowie die zugehörige Skala und passt die Größe der einzelnen Elemente dynamisch entsprechend den anzuzeigenden Werten an.

2.3.23 SL_Plotter

Die Klasse SL_Plotter implementiert ein in SL_Window integrierbares UI-Modul zur Darstellung von Kurvenverläufen zur Auswertung von Simulationsergebnissen. Als Datenquellen für die einzelnen angezeigten Kurven kommen gepufferte SL_DiagramData-Objekte zum Einsatz. Die Anzeige der Achsen und Skalen erfolgt automatisch und richtet sich nach dem dargestellten Wertebereich. Für die Berechnung und graphische Ausgabe kommen die Hilfsklassen Plotter, Axis, XAxis und YAxis zum Einsatz.

2.3.24 SL_EventController

Die Klasse SL_EventController wird zur Steuerung der Simulation benötigt, wobei zu jedem Zeitpunkt lediglich eine Instanz dieser Klasse existieren darf. Es werden Methoden zum Registrieren verschiedener Simulationskomponenten wie SL_Provider-Objekte oder Datenquellen-Objekten (SL_Queue und SL_Table) zur Verfügung gestellt. Der Zugriff auf diese registrierten Objekte erfolgt anhand ihres Namens. Weiterhin lassen sich Stationen registrieren, die bei Eintritt von Ereignissen (beispielsweise Start, Stop, SpeedChange) benachrichtigt werden, um ihrerseits Aktionen durchzuführen.

Der EventController beinhaltet verschiedene 'Tabpages', welche die Steuerung der Simulationsgeschwindigkeit ermöglichen (siehe TabPageClock), registrierte Timerevent-Objekte anzeigen (siehe TabPageEventList), registrierte SL_Provider-Objekte mit

zugewiesenem Namen auflisten (siehe TabPageServiceData) und eingetragene Datenquellen wie SL_Counter-Objekte für die statistische Auswertung mit den aktuellen Zählerständen anzeigen (siehe TabPageDataCollection). Die einzelnen 'Tabpages' und die enthaltenen Informationen werden während eines Simulationslaufs in einem festgelegten Zeitintervall ständig erneuert.

Die Klasse TabPageClock stellt Funktionen zur Steuerung der Simulation zur Verfügung. Hier kann eine Simulations-Endzeit eingestellt, die Geschwindigkeit der Simulation (Zeitraffer und Zeitlupe) per Schieberegler verändert sowie die Simulation gestoppt und auf den Startzustand zurückgesetzt werden. Weiterhin lässt sich die Animation der MEs abschalten. Eine weitere Aufgabe dieser Klasse ist das Registrieren von Simulationsobjekten, welche bei Eintritt bestimmter Ereignisse benachrichtigt werden, um ihrerseits eine Funktion auszuführen. So werden beispielsweise die SL_Counter-Objekte bei Betätigung des Reset-Schalters ('Reset-Event') auf den initialen Zählerwert zurückgesetzt. Weiterhin ist es möglich, verschiedene Objekte nach Ablauf eines Zeitintervalls zu benachrichtigen (beispielsweise SL_Source), um in einem voreingestellten Intervall neue MEs zu erzeugen. Diese Funktion übernimmt die Klasse SL_ObjectNotifier, welche die action()-Methode eines GUI-Objekts nach einem zuvor registrierten Zeitintervall aufruft.

Die Klasse TabPageDataCollection dient der graphischen Darstellung von registrierten Datenquellen-Objekten (bspw. SL_Counter). Während eines Simulationslaufs erscheinen hier die aktuellen Werte, welche in einer Tabelle (siehe TableModel_DataCollection) aufgelistet werden. Die hier vorliegenden Werte können für statistische Auswertungen herangezogen werden.

Die Klasse TabPageEventList dient der graphischen Darstellung der in TabPageClock registrierten Objekte, welche bei Eintritt eines Timer-Ereignisses benachrichtigt werden sollen. Die graphische Darstellung enthält einerseits den Namen des registrierten Objekts sowie die aktuell zugewiesene Benachrichtigungszeit in der Darstellung 'hh:mm:ss'.

Die Klasse TabPageServiceData hält registrierte SL_Provider-Objekte in einer Hashtabelle und stellt Methoden zum Zugriff auf diese Objekte zur Verfügung. Weiterhin werden die Hashtabellen-Elemente innerhalb einer Tabpage in der Simulationssteuerung (SL_EventController) graphisch angezeigt.

2.3.25 SL_Window

Die Klasse SL_Window stellt ein Fenster zur Verfügung, das einerseits einen Objekt-Browser (siehe SL_Browser) beinhaltet, andererseits über eine virtuelle Gitterstruktur verfügt, in welche die für die Simulation benötigten GUI-Komponenten seitens des Benutzers eingetragen werden. Das Einfügen beziehungsweise Verschieben von GUI-Komponenten wird durch einen Dialog der Klasse DialogInsertUIModule realisiert. Der Name des Fensters und der Elternknoten für die Darstellung im Objekt-Browser werden beim Erzeugen des SL_Window-Objekts (mittels getInstance()) als Parameter übergeben.

2.3.26 SL_Browser

SL_Browser-Objekte dienen als Bestandteil aller SL_Window-Objekte der Visualisierung und Benutzersteuerung der Simulations-Konfiguration. Das durch SL_BrowserIcon verwaltete MutableTreeNode-Objekt bildet den Knoten bzw. das Blatt in der Browser-Baumstruktur. Die Abhängigkeiten der einzelnen Simulations-Objekte werden automatisch im Browser angezeigt und aktualisiert. Der Browser zeigt jeweils den Teil der Simulations-Objekthierarchie an, der vom zugehörigen SL_Window verwaltet wird. Der Konstruktor (bzw. die getInstance()-Methode) von SL_Browser erwartet als Parameter eine Referenz auf das zu diesem Browser gehörende SL_Window-Objekt.

2.3.27 DialogStationConfig

DialogStationConfig implementiert einen (modalen) Dialog, der die Grundlage aller Konfigurationsdialoge bildet. Kernbestandteil ist ein JTabbedPane, dem beliebige Seiten (von Component abgeleitet; verwaltet von 'DlgPage...'-Objekten) mittels der Methode addTab() hinzugefügt werden können, die spezifische Elemente zur Konfiguration von Stationen usw. enthalten. Weiterhin werden die Buttons 'OK' und 'Cancel' bereitgestellt. Die Auswertung der Benutzer-Eingaben auf den verschiedenen Seiten wird durch die den Dialog generierende Instanz gewährleistet.

ConfPnlSource implementiert ein Konfigurations-Panel, das als Bestandteil einer DlgPagePropertyEditor-Konfigurationsseite im DialogStationConfig-Dialog angezeigt wird. Es dient der Modifizierung des zeitlichen Abstands, in dem MoveableEntity-Objekte durch eine SL_Source erzeugt werden. Der einzige Konstruktor von ConfPnlSource erwartet als Parameter das bisherige Intervall der SL_Source. Beim Schließen des Dialogs kann der aktuelle Wert durch die Methode getInterval() abgefragt werden.

ConfPnlDataStation implementiert ein Konfigurations-Panel, das als Bestandteil einer DlgPagePropertyEditor-Konfigurationsseite im DialogStationConfig-Dialog angezeigt wird. Es dient der Auswahl der Datenquelle für ein SL_DataStation-Objekt durch den Benutzer. Der einzige Konstruktor von ConfPnlDataStation erwartet als Parameter eine Referenz auf die momentan aktuelle Datenquelle. Diese kann nachträglich durch die Methode setChosenObject() ausgewählt werden. Beim Schließen des Dialogs lässt sich die aktuell ausgewählte Datenquelle durch die Methode getChosenObject() abfragen. Die Liste aller vorhandenen Datenquellen wird in Form eines Vector-Objekts, das durch die Methode getDataModuleVector() der Klasse SL_Data ermittelt werden kann, zur Verfügung gestellt.

DlgPageUserCodeEditor implementiert eine Konfigurationsseite für SL_Station-Objekte und alle abgeleiteten Klassen, die in einen Konfigurationsdialog (DialogStationConfig) eingefügt wird.
Diese Seite enthält GUI-Elemente, mit denen der Benutzer den Usercode der Station bearbeiten kann. Der einzige Konstruktor von DlgPageUserCodeEditor erwartet als Parameter einen String mit dem bisherigen Usercode. Alle Änderungen müssen nach Dialogbeendigung mit der entsprechenden Methode (getUserCode()) abgefragt und von der den Dialog erzeugenden Instanz bearbeitet werden.

DlgPagePropertyEditor implementiert eine Konfigurationsseite für SL_Station-Objekte und alle abgeleiteten Klassen, die in einen Konfigurationsdialog (DialogStationConfig) eingefügt wird. Diese Seite enthält GUI-Elemente, mit denen der Benutzer die Ausgänge sowie den Deklarationsteil des Usercode der zugehörigen Station konfigurieren kann. Der einzige Konstruktor von DlgPagePropertyEditor erwartet als Parameter einen Vector mit den Namen der bisherigen Stations-Ausgänge, die bisherigen Usercode-Deklarationsteil sowie ggf. ein Panel mit weiteren GUI-Objekten zur Konfiguration spezieller Parameter. Das Panel wird als Bestandteil dieser Konfigurationsseite angezeigt. Alle Änderungen müssen nach

Dialogbeendigung mit den entsprechenden Methoden (getExitNames(), getUserDeclarations() und ggf. den Methoden des Zusatz-Panels) abgefragt und von der den Dialog erzeugenden Instanz bearbeitet werden.

DlgPageMultiNetFrameConfig implementiert eine Konfigurationsseite für SL_MultiNetFrame-Objekte, die in einen Konfigurationsdialog (DialogStationConfig) eingefügt wird. Diese Seite enthält GUI-Elemente, mit denen der Benutzer die Anzahl der Simulations-Netz-Instanzen von SL_MultiNetFrame in verschiedenen Simulations-Zeiträumen bestimmen kann. Der einzige Konstruktor von DlgPageMultiNetFrameConfig erwartet als Parameter eine Referenz auf ein TimeTableModel-Objekt, das zum einen als Daten-Modell für die hier verwaltete Tabelle, zum anderen zur Speicherung der Konfigurationsdaten innerhalb SL_MultiNetFrame dient. Alle Benutzereingaben wirken sich somit unmittelbar auf das Verhalten des zugehörigen SL_MultiNetFrame aus. Eine Verarbeitung nach Schließen des Dialogs durch die den Dialog aufrufende Instanz ist nicht erforderlich.

DlgPageDiagramPropertyEditor implementiert eine Konfigurationsseite für SL_Diagramm-Objekte, die in einen Konfigurationsdialog (DialogStationConfig) eingefügt wird. Diese Seite enthält GUI-Elemente, mit denen der Benutzer Datenquellen (SL_DiagramData) für Diagramme erzeugen, löschen und konfigurieren sowie die Abfrageintervalle einstellen kann. Der einzige Konstruktor von DlgPageDiagramPropertyEditor erwartet als Parameter eine Referenz auf die Liste der bereits vorhandenen Datenquellen sowie die bisherigen Abfrageintervalle. Alle Änderungen bzgl. Datenquellen sowie der Abfrageintervalle müssen nach Dialogbeendigung mit den entsprechenden Methoden (getDiagramDataObjects(), getDataRefreshInterval() und getRedrawInterval()) abgefragt und von der den Dialog erzeugenden Instanz bearbeitet werden.

Die Klasse DlgPageDBAccess implementiert eine Konfigurationsseite für SL_Table-Objekte, die in einen Konfigurationsdialog (DialogStationConfig) eingefügt wird. Über diesen Dialog erstellt der Benutzer eine Referenz auf eine Datenquelle vom Typ DBSqlAccess beziehungsweise DBFileAccess. Nachdem eine Zuweisung erfolgreich durchgeführt wurde, erscheint im unteren Bereich des Dialogs eine Tabelle mit den Spaltennamen der Datenquelle, dem zugewiesenen Primärschlüssel sowie den Spaltennamen, die innerhalb der Simulation zu benutzen sind. Die Primärschlüssel-Position sowie die Simulations-Spaltennamen können hier vom Benutzer konfiguriert werden.

2.3.28 DialogInsertUIModule

DialogInsertUIModule implementiert einen (modalen) Dialog, mit dessen Hilfe der Benutzer auswählen kann, in welches SL_Window-Fenster sowie an welche Raster-Position innerhalb des Fensters ein UI-Modul eingefügt werden soll. Der einzige Konstruktor von DialogInsertUIModule erwartet als Parameter neben dem anzuordnenden UI-Modul auch einen Vektor aller zur Verfügung stehenden Fenster und ggf. den alten "Besitzer" des UI-Moduls. Die durch den Benutzer initiierten Aktionen (Erzeugen eines neuen Fensters oder UI-Modul an neue Raster-Position setzen) werden unmittelbar innerhalb von DialogInsertUIModule ausgeführt, d. h. ein Abfragen und Auswerten der Eingaben bei der den Dialog generierenden Instanz sind nicht nötig.

2.3.29 DialogComboBox

Die Klasse DialogComboBox stellt einen Benutzerdialog für die Eingabe von Zeichenketten und numerischen Werten zur Verfügung. Dabei können dem Benutzer Werte vorgegeben werden, die dieser auswählen und gegebenenfalls editieren kann. Objekte vom Typ DialogComboBox werden beispielsweise zum Einstellen der queueSize eines Konnektors verwendet.

2.3.30 SL_ClassRegistry

Die Klasse SL_ClassRegistry stellt Methoden zur Verfügung, die das Abspeichern und Einladen von (Simulations-) Konfigurationen unterstützen und steuern. Für das Erzeugen eines Konfigurationsbeschreibungs-Strings, der in einer Konfigurations-Datei abgespeichert werden kann, existieren Klassenmethoden (getTupleString()), die Teil-Beschreibungen für einfache Datentypen und verschiedenen Objekt-Typen generieren. Beim Einladen einer Konfiguration wird eine Instanz von SL_ClassRegistry erzeugt, die Methoden zur Auswertung des Beschreibungs-Strings (get...FromDescrString()), zur Erzeugung der (GUI- und Simulations-) Objekte (createObject()), zum Registrieren der erzeugten Objekte (registerObject()) und Benutzercode-Sonderbehandlung (registerUserCodeCompileReq() und registerUserCodeReq()) sowie zum Kompilieren des Benutzer-Sourcecode

(compileUserCode()) bereitstellt und den Ablauf der Ladevorgang-Nachbearbeitung steuert (linkObjects()).

Die Ladesequenz läuft in folgenden Schritten ab:
1. Erzeugen und Konfigurieren der Objekte entsprechend der Konfigurationsbeschreibung (durch entsprechenden Konstruktor). Die Abarbeitung des Konfigurationsbaumes erfolgt in Tiefensuche. Registrieren aller geladenen Objekte über ihre Object ID durch Methode registerObject() des SL_ClassRegistry-Objekts.
2. Aufruf der Methode initObjLinks() für alle erzeugten Objekte, um erstens die Abhängigkeiten zwischen den Objekten aufzulösen (da zu diesem Zeitpunkt alle Objekte erzeugt wurden, können jetzt die in der Konfigurationsbeschreibung enthaltenen Object IDs durch die Referenzen auf die entsprechenden Objekte ersetzt werden) und zweitens den Benutzer-Sourcecode für Stationen und Diagramm-Datenquellen zu generieren. Die Quelldateien werden mittels registerUserCodeCompileReq(), die zugehörigen Objekte über registerUserCodeReq() beim SL_ClassRegistry-Objekts registriert.
3. Compilierung der in Schritt 2 erzeugten und registrierten Benutzercode-Quelldateien. Dazu erzeugt SL_ClassRegistry eine Instanz der Klasse SunJavaCompiler, die die Schnittstelle zum Sun-JAVA-Compiler bildet.
4. Aufruf der Methode loadUserCode() für alle durch registerUserCodeReq() registrierten Objekte, um den kompilierten Benutzercode zu laden.

Zur Ausgabe von Fehlermeldungen (beim Compilieren des Benutzercodes) bedient sich SL_ClassRegistry der Klasse: SL_InfoWindow. Diese ermöglicht die Ausgabe von Textstrings in einem separaten Fenster an.

2.3.31 SL_ObjectCounter

SL_ObjectCounter stellt einen Objekt-Zähler zur Verfügung, der für die Generierung eindeutiger Objekt IDs verantwortlich ist. Von dieser Klasse existiert jeweils nur eine Instanz (singleton).

2.3.32 SL_Utilities

Die Klasse SL_Utilities stellt verschiedene Hilfsmethoden zur Verfügung, welche von allen Klassen der Simulationsbibliothek benutzt werden können.

2.3.33 SL_MoveableEntity

Die Klasse SL_MoveableEntity (kurz: ME) repräsentiert ein bewegliches Datenobjekt, das zwischen einzelnen Bearbeitungsstationen weitergereicht wird. Diese Klasse implementiert das Interface SL_MEDataInterface und enthält eine Hashtabelle, in der die zu transportierenden Daten gespeichert werden. Verschiedene Methoden ermöglichen das Hinzufügen (bspw. putInt()), Auslesen (bspw. getBool()) und Entfernen (bspw. removeObject()) von (zu transportierenden) Einträgen.

2.4 Benutzungsanleitung

Nach dem Start von EnSiCC besteht die Möglichkeit, über das Menü des Programmfensters eine vorhandene Konfiguration zu laden (Eintrag „File | Load“) oder eine neue anzulegen (Menü „File | New“). In letzterem Fall wird ein erstes Simulationsfenster erzeugt, das den automatisch generierten EventController beinhaltet. Mittels Kontextmenüs der im Objektbrowser angezeigten Fenster-Elemente können weitere Fenster („New Window“) sowie Netzwerke („Add Network“), Mehrfach-Netzwerke („Add Multi-Network“), Datenmodule („Add Data-Module“) und Diagramme („Add Diagram“) erstellt werden.

Das Positionieren der oben genannten UI-Komponenten unterstützt der in Abbildung 7 dargestellte Dialog. Dieser wird beim Neuerzeugen automatisch aktiviert, kann aber zu späteren Zeitpunkten mittels Kontextmenüeintrag („New Grid Position“) zur Umpositionierung von Komponenten aufgerufen werden.

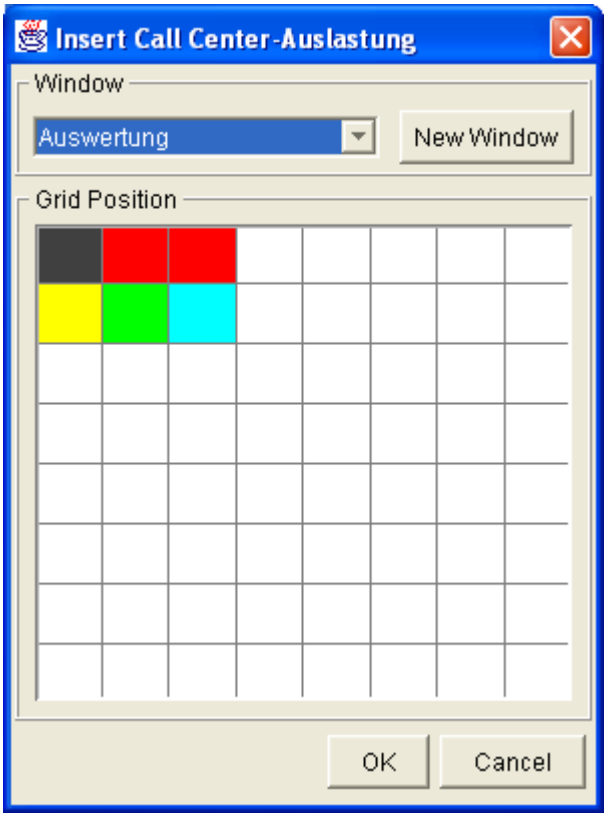


Abbildung 7 Dialog zum Einfügen von UI-Komponenten

Der Dialog zeigt das virtuelle Raster des korrespondierenden Fensters an. Dieses visualisiert durch unterschiedliche Farben die Verteilung der bereits existierenden UI-Komponenten. Falls eine Komponente verschoben werden soll, ist die bisherige Position schwarz dargestellt. Die neue Position wird durch Aufspannen eines Rechtecks auf dem Raster mit der Maus festgelegt. Bei Bedarf ermöglicht dieser Dialog auch das Erzeugen eines neuen Fensters.

Das Hinzufügen von Objekten und Konnektoren zu einem Simulationsnetzwerk ermöglicht dessen Kontextmenü. Diese Simulations-Objekte können auf der Arbeitsfläche per Drag & Drop frei verschoben und durch objektspezifische Kontextmenüs konfiguriert oder gelöscht werden.

Beim Erzeugen eines neuen Konnektors ist dessen Anfangspunkt (blauer Anfasser) auf das gewünschte Quell-Objekt zu positionieren und anschließend per Mausklick zu bestätigen. Der Endpunkt (grüner Anfasser) wird auf das Ziel-Objekt ausgerichtet und dort fixiert. Durch Markieren des Konnektors mit der Maus ist ein Verschieben über die nun aktiven Anfasser möglich. Sobald der Anfangspunkt eines Konnektors an ein Objekt angedockt wird, öffnet sich bei diesem eine Auswahlbox, mit der ein Ausgang des Objekts als Quelle für den Konnektor gewählt werden kann (sofern mehrere Ausgänge definiert wurden).

Den Status des Konnektors zeigt dessen Farbe an: Bei einem schwarzen Konnektor sind sowohl Anfangs- als auch Endpunkt mit Objekten assoziiert. Dagegen deutet ein roter Konnektor auf eine unvollständige Einbindung hin.

Die Namen aller Komponenten lassen sich durch Markieren im Objektbrowser und Drücken von „F2“ oder (bei Simulationsobjekten) über Kontextmenüeintrag („Configure“) ändern.

Weiterhin ist es jederzeit möglich, Änderungen an der Simulationskonfiguration über das Menü des Programmfensters (Eintrag „File | Save“) abzuspeichern.

2.4.1 Kontextmenüeinträge

Tabelle 1 enthält die in der Simulationsumgebung vorkommenden Kontextmenüeinträge und die Beschreibung der einzelnen Befehle.

New Grid Position	Ermöglicht das Zuweisen einer Position innerhalb der Gitterstruktur des aktuellen Fensters.
Delete	Löscht das ausgewählte GUI-Objekt.
Configure	Öffnet einen Dialog zur Konfiguration der ausgewählten Komponente.
New Window	Erzeugt ein neues Fenster.
Hide Window / Show Window	Blendet ein existierendes Fenster ein bzw. aus.
Optimize Width	Optimiert die Darstellungsbreite des Objektbrowsers entsprechend der längsten dargestellten Komponentenbezeichnung.
Add Station	Hinzufügen einer Bearbeitungsstation in das ausgewählte Netzwerk.
Add Source	Hinzufügen einer Source-Station in das ausgewählte Netzwerk.
Add Datasink	Hinzufügen einer Datensenke-Station in das ausgewählte Netzwerk.
Add Counter	Hinzufügen einer Zählstation in das ausgewählte Netzwerk.
Add Variable	Hinzufügen einer Variablen in das ausgewählte Netzwerk.
Add Datastation	Hinzufügen einer Datenzugriffs-Station in das ausgewählte Netzwerk.
Add Service | Add Consumer	Hinzufügen einer Consumer-Station in das ausgewählte Netzwerk.
Add Service | Add Provider	Hinzufügen einer Provider-Station in das ausgewählte Netzwerk.
Add Connector	Hinzufügen eines Konnektors in das ausgewählte Netzwerk.
Add Network	Hinzufügen eines Netzwerks.
Add Multi-Network	Hinzufügen eines Multi-Netzwerks.
Add Datamodule | Add Queue	Hinzufügen einer Warteschlange.

Add Datamodule | Add Table | Add SQL-Database	Hinzufügen einer SQL-Datenbank.
Add Datamodule | Add Table | Add File-Database	Hinzufügen einer Datei-Datenbank.
Add Diagram | Add Chart	Hinzufügen eines Balken-Diagramms.
Add Diagram | Add Plotter	Hinzufügen eines Plotter-Diagramms.

Tabelle 1 Kontextmenüeinträge

2.4.2 Konfiguration von Stationen

Abbildung 8 zeigt den Konfigurationsdialog für Stationen. Der Benutzer kann das Verhalten von Stationen über Benutzercode konfigurieren. Es handelt sich hierbei um das Erstellen von JAVA-Benutzercode-Klassen, die zur Laufzeit kompiliert und zu den bestehenden Stations-Objekten hinzugefügt werden.

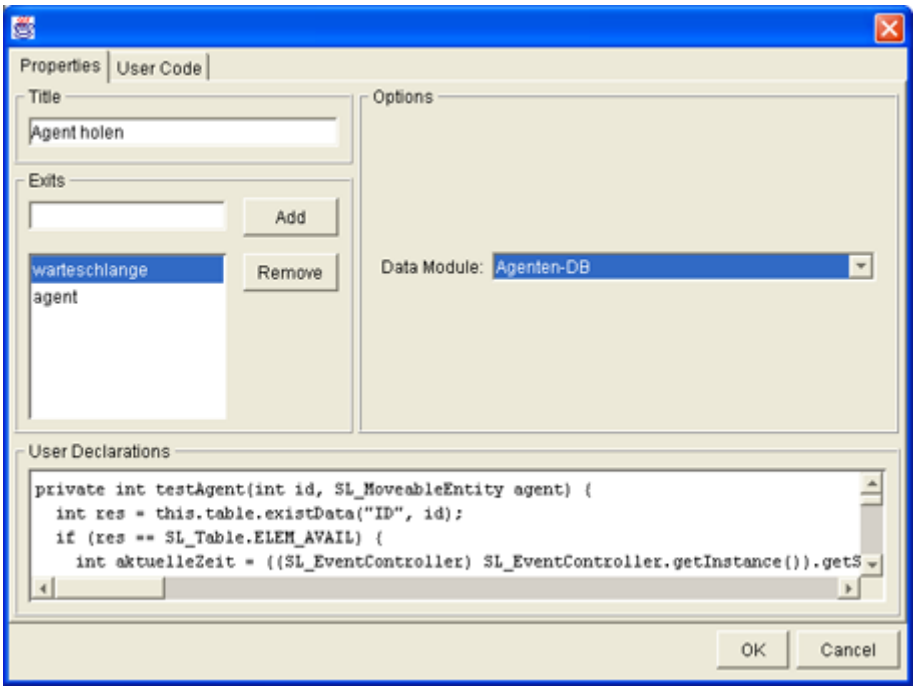


Abbildung 8 Konfigurationsdialog für Stationen

Der Benutzer hat auf zwei Bereiche der JAVA-Benutzercode-Klasse Zugriff: Zum einen kann er über „User Declarations“ Methoden und Felder deklarieren, zum anderen über „User Code“ die vordefinierte Methode „void **userCode**()“ mit Inhalt füllen.
Die Abarbeitung des Benutzercodes umfasst folgende Schritte:

1. Wenn die Station an einem eingehenden Konnektor ein ME vorfindet, wird dieses übernommen und dem Benutzercode-Objekt zur Bearbeitung übergeben.
2. Das Benutzercode-Objekt legt das übergebene ME in der Variablen **me** ab und ruft die benutzerdefinierte Methode **userCode**() auf, die die weitere Bearbeitung übernimmt.
3. Nach abgeschlossener Bearbeitung muss **me** durch den Benutzercode mit der Methode **sendME**() an einen ausgehenden Konnektor übergeben werden. Diese Konnektoren werden über Namen angesprochen, die zuvor durch den entsprechenden Benutzerdialog definiert worden sein müssen (siehe Abbildung 8).

Neben der gesamten Funktionalität, die das JDK zur Verfügung stellt, können im Benutzercode einige spezielle Methoden und Variablen verwendet werden. Eine Übersicht bietet folgende Zusammenstellung.

(i) Felder und Methoden für SL_Station (und alle abgeleiteten Objekte)

protected SL_MoveableEntity **me**

Diese Variable speichert das zur Bearbeitung an den Benutzercode übergebene ME.

protected boolean **sendME**(String *exitName*, SL_MoveableEntity *me*)

Sendet *me* an den Ausgang mit dem Namen *exitName*. Wenn das ursprünglich zur Bearbeitung übergebene ME dem übergebenen ME entspricht, wird dieses nach dem Senden aus dem Benutzercode-Objekt entfernt.

protected boolean **isInterrupted**()

Testet, ob eine Anforderung zum Beenden des Benutzercodes vorliegt. Falls true zurückgegeben wird, sollte die Bearbeitung schnellstmöglich abgebrochen werden.

protected void **sleep**(long *millis*)

Unterbricht die Ausführungs-Thread des Benutzercodes für *millis* Millisekunden.

protected int **getSimulationTime**()

Liefert die aktuelle Simulationszeit (in Sekunden).

(ii) Spezielle Felder für SL_DataStation

Je nach zugeordnetem Datenmodul kann dieses über

private SL_Queue **queue**

oder

private SL_Table **table**

angesprochen werden.

(iii) Spezielle Methoden für SL_Variable

public void **setVariable**(double *variable*)

Setzt die verwaltete Double-Variable auf den Wert *variable*.

public double **getVariable**()

Liefert den Wert der verwalteten Double-Variable.

(iv) Spezielle Methoden für SL_Provider

protected void **transmitME**(SL_MoveableEntity *me*)

Sendet *me* an den Consumer, der das ME ursprünglich an diesen Provider übermittelt hat, zurück.

protected void **lockProvider**()

Sperrt den Provider, d. h. es können keine weiteren MEs an diesen Provider übergeben werden.

protected void **unlockProvider**()

Hebt die Sperrung des Providers wieder auf.

protected boolean **isProviderLocked**()

Testet, ob der Provider gesperrt ist.

(v) Spezielle Methoden für SL_Consumer

protected SL_Provider **getProviderByName**(String *providerName*)

Liefert den Provider mit dem exakten Namen *providerName* (ohne Beachtung der Groß-/Kleinschreibung). Falls kein Provider mit diesem Namen existiert, wird null zurückgegeben. Diese Methode steht nur im „Input User Code“ des Consumers zur Verfügung.

protected SL_Provider **getProviderByPartOfName**(String *providerName*)

Liefert den Provider, dessen Name den Bestandteil *providerName* (ohne Beachtung der Groß-/Kleinschreibung) aufweist. Falls kein solcher Provider existiert, wird null zurückgegeben. Diese Methode steht nur im „Input User Code“ des Consumers zur Verfügung.

protected void **transmitME**(SL_MoveableEntity *me*, SL_Provider *p*)

Übergibt *me* an den Provider *p*. Diese Methode steht nur im „Input User Code“ des Consumers zur Verfügung. Diese Methode steht nur im „Input User Code“ des Consumers zur Verfügung.

protected boolean **isProviderLocked**(SL_Provider *p*)

Testet, ob der Provider *p* gesperrt ist. Wenn false zurückgegeben wird, wird ein Aufruf von **transmitME** scheitern.

Um auf die in einer ME gespeicherten Daten zuzugreifen, stehen Methoden zum

- Auslesen (public <JavaType> **get<Type>**(String *name*)),
- Einfügen (public void **put<Type>**(String *name* , <JavaType> *value*)) und
- Löschen (public void **remove<Type>**(String *name*))

zur Verfügung.

Dabei müssen die verschiedenen Datentypen <JavaType> und **<Type>** durch folgende Einträge ersetzt werden:

<JavaType>	<Type>
boolean	Bool
double	Double
int	Int
long	Long
SL_MoveableEntity	ME
Object	Object
String	String

Tabelle 2 JAVA-Datentypenbezeichnung in MEs

Um die Eingabe des Benutzercodes zu erleichtern, lässt sich durch einen Klick mit der rechten Maus-Taste auf den Editor ein Kontext-Menü aufrufen, das eine Auswahl von häufig benötigten Code-Segmenten bietet, die bei Auswahl an die momentane Cursor-Position in den Benutzercode eingefügt werden (siehe Abbildung 9). Die durch „#“ gekennzeichneten Passagen brauchen nur noch durch die korrekten Werte ersetzt zu werden.

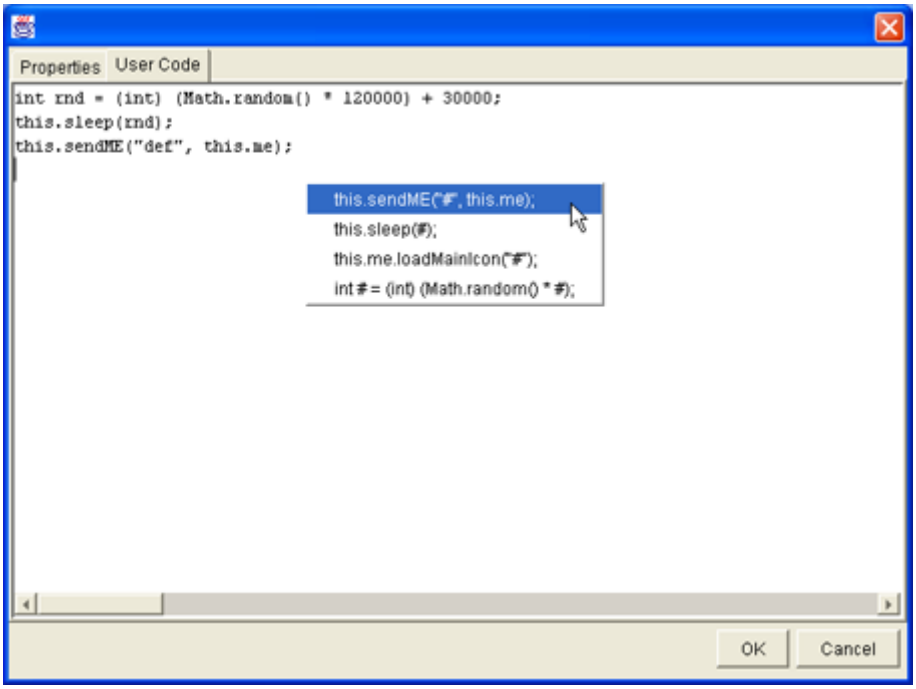


Abbildung 9 Benutzercode-Dialog für Stationen

2.4.3 Konfiguration von Table-Objekten

Die Konfiguration eines Table-Objekts geschieht durch einen Benutzerdialog (siehe Abbildung 10). Hier kann einerseits mittels eines Datei-Dialogs eine Datendatei als Grundlage für eine File-Database ausgewählt werden. Für eine SQL-Database sind andererseits der Name der ODBC-Verbindung (registriert über ODBC-Admin unter Windows) sowie der Name der zugrundeliegenden Tabelle einzugeben. Nach Auswahl einer entsprechenden Datenquelle werden die Spaltennamen in der rechten Tabellenspalte des Dialogs angezeigt. Die mittlere Spalte dient zur Auswahl des Primärschlüssels. In der linken Spalte kann der Benutzer eigene Spaltennamen angeben, auf welche innerhalb einer durchzuführenden Simulation zugegriffen wird. Das Editieren geschieht durch Maus-Doppelklick auf den zu ändernden Spaltennamen. Nach Bestätigen des „OK“-Buttons ist die Datenquelle in der Simulationsumgebung eingetragen.

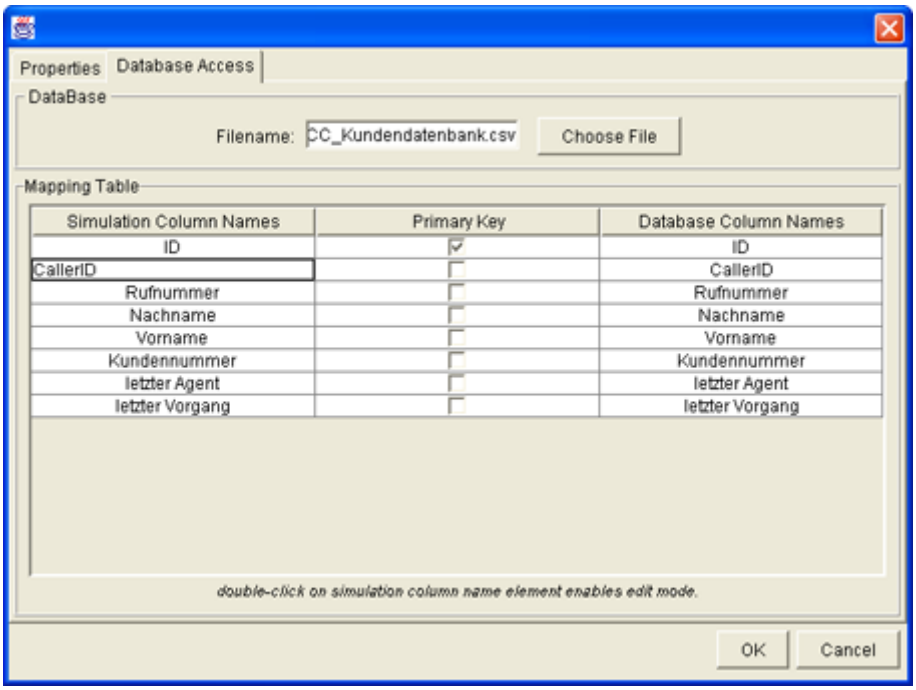


Abbildung 10 Konfigurationsdialog für Table-Objekte

Folgendes ist bei Anbindung einer externen Datenquelle zu beachten:
SQL-Datenbank: Primärschlüssel (Integer) muss vorhanden sein. Alle anderen Tabellenspalten enthalten String-Werte.
File-Datenbank: Erste Zeile enthält Spaltenüberschriften (durch Semikolon getrennt, kein abschließendes Zeilen-Semikolon). Ein Primärschlüssel (bspw. ID) muss vorhanden sein.

2.4.4 Konfiguration von Plotter und Chart

Abbildung 11 zeigt den Konfigurationsdialog für die Diagrammtypen Plotter und Chart.

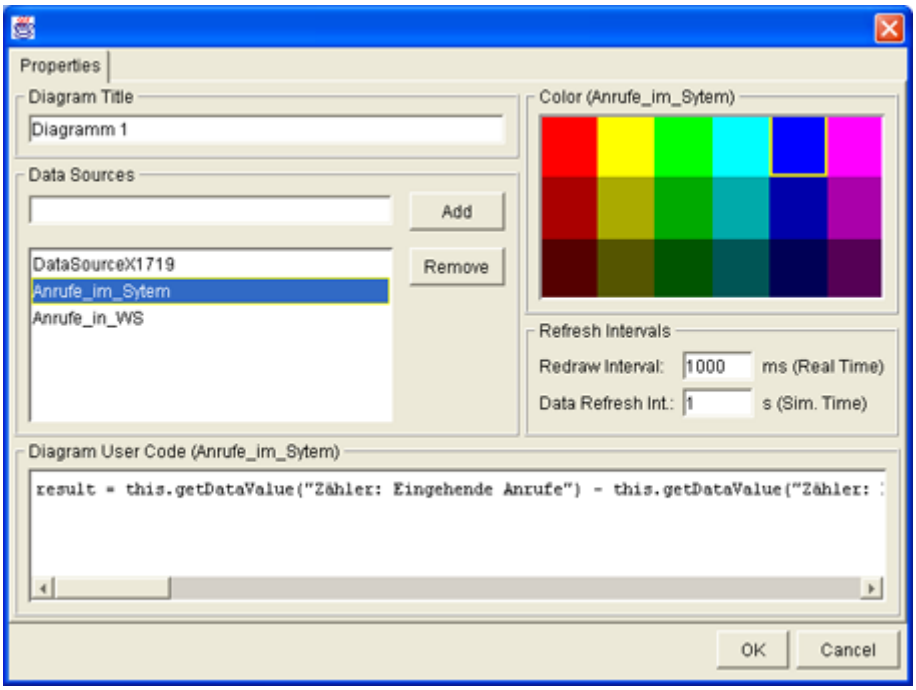


Abbildung 11 Konfigurationsdialog für Diagramme

Jedem Diagramm-Element (Balken bzw. Kurve) entspricht eine Diagramm-Datenquelle, die im Dialog-Feld „Data Sources“ hinzugefügt oder entfernt werden kann.

Eine Datenquelle für ein Diagramm-Element kann durch den Benutzer bezüglich angezeigtem Wert und Darstellungsfarbe konfiguriert werden. Dazu muss der Benutzer eine Datenquelle aus der Liste (durch Anklicken) auswählen. Die Berechnung des darzustellenden Wertes erfolgt – ähnlich wie die Konfiguration von Stationen – durch JAVA-Benutzercode, indem die Methode

public double **getValue**(double *x*)

der Klasse SL_DiagramData ausprogrammiert wird. Diese Methode wird durch das Diagramm-Objekt in regelmäßigen Abständen aufgerufen. Der übergebene Parameter *x* enthält den X-Wert, für den die Berechnung (des zugehörigen Y-Wertes) ausgeführt werden muss. Bei Chart-Diagrammen ist *x* immer 0, bei Plotter-Diagrammen dagegen wird *x* durch eine ausgezeichnete Datenquelle („DataSourceX...“) berechnet, die beim Erzeugen eines Plotter-Objekts automatisch eingefügt wird und standardmäßig die aktuelle Simulationszeit

liefert. Der berechnete Wert darf nicht mit return zurückgegeben, sondern muss der double-Variablen **result** zugewiesen werden!

Für den Zugriff auf die Datenquellen in den Simulationsnetzen (SL_Counter-, SL_Variable- und SL_Queue-Objekte) kann im Benutzercode die Methode

protected double **getDataValue**(String *counterName*)

herangezogen werden, wobei *counterName* dem Namen der abzufragenden Datenquelle entsprechen muss (ohne Berücksichtigung der Groß-/Kleinschreibung). Eine Liste aller zur Verfügung stehenden Netzwerk-Datenquellen lässt sich über das Kontextmenü des Benutzercode-Editors abrufen.
Die aktuelle Simulationszeit liefert die Methode

protected int **getSimulationTime**().

Das Zeitintervall (in Sekunden Simulationszeit) zwischen zwei Ausführungen der Methode **getValue**() aller Datenquellen eines Diagramms wird durch „Data Refresh Int.“ bestimmt. „Redraw Interval“ legt den zeitlichen Abstand (in Sekunden Realzeit) zwischen den Aktualisierungen der graphischen Darstellung des Diagramms fest. Die Unterscheidung zwischen „Data Refresh Int.“ und „Redraw Interval“ ist für gepufferte Datenquellen, die bei Plottern zum Einsatz kommen, relevant.

3. Simulation eines Call Centers

Zur Demonstration der entwickelten Simulationsumgebung wird im folgenden Kapitel das Modell eines Inbound-Call Centers vorgestellt. Die EnSiCC-Umgebung ermöglicht das Erstellen weitaus komplexerer Modelle, jedoch soll folgende Darstellung ausreichen, um die Leistungsfähigkeit der Simulationsumgebung zu verdeutlichen und die zur Simulation benötigten Komponenten und deren Arbeitsweise zu erläutern.

3.1 Aufbau und Ablauf

Anhand des in Abbildung 12 dargestellten Call Center-Modells soll der Aufbau und Ablauf einer Simulation mit den daran beteiligten Bearbeitungsstationen vorgestellt werden. Die Abbildung zeigt die Fenster Welt, Call Center und Agenten. Das Fenster Welt beinhaltet ein Simulationsnetzwerk, welches für die Erzeugung von Anrufen zuständig ist und die Menge der potentiellen Kunden des Call Centers repräsentiert. Das Call Center-Fenster verwaltet zwei Netzwerke: die TK-Anlage und das ausgelagerte Multi-Netframe-Fenster Agenten zur Darstellung des Agentennetzwerks.

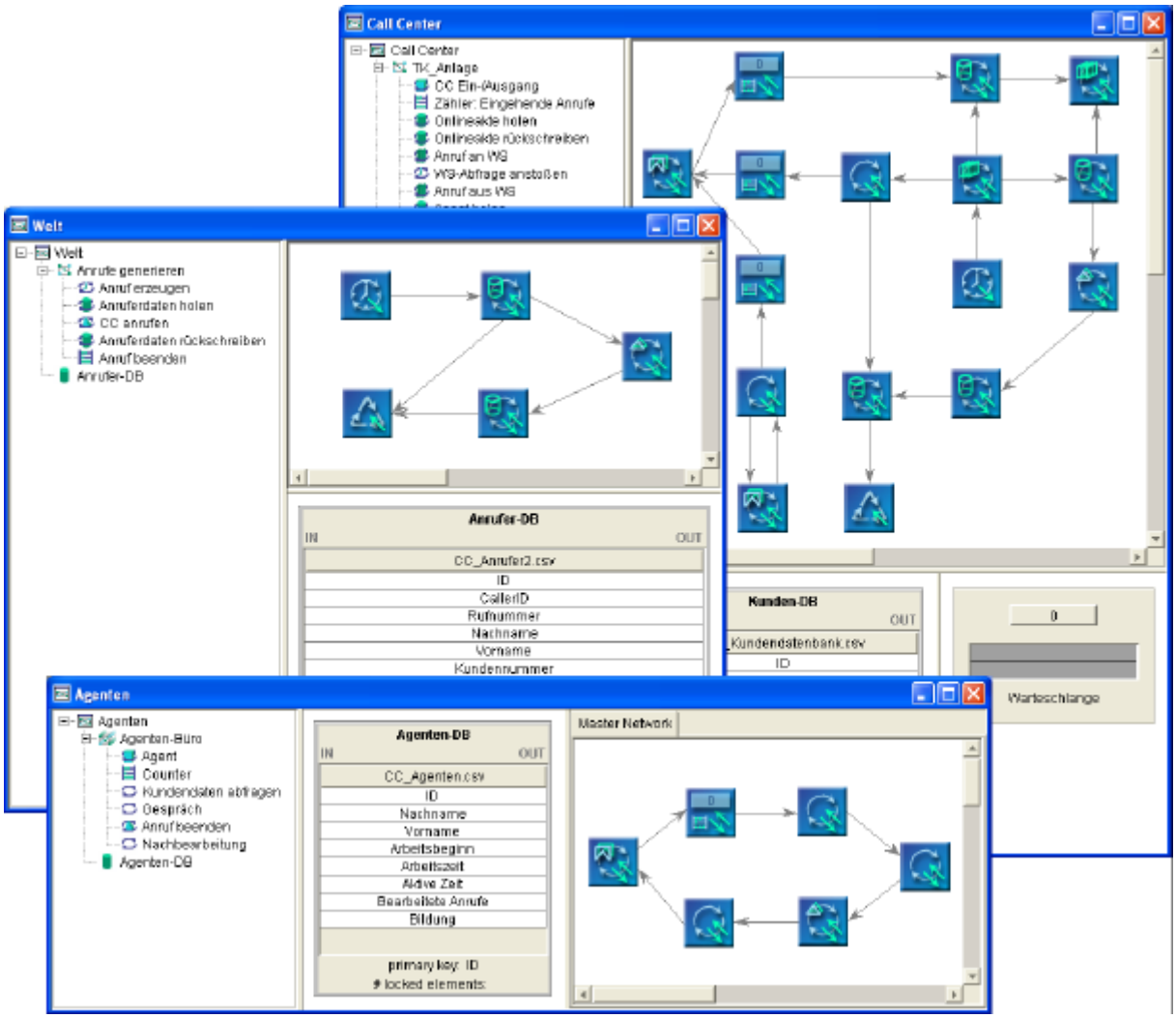


Abbildung 12 Call Center-Modell (Ausschnitt)

Eine Übersicht über die Hierarchie der Simulationskonfiguration liefert der Objektbrowser des Fensters Simulations-Steuerung. Dieses enthält ebenfalls den EventController, der zur Steuerung des Simulationsablaufs dient (siehe Abbildung 13).

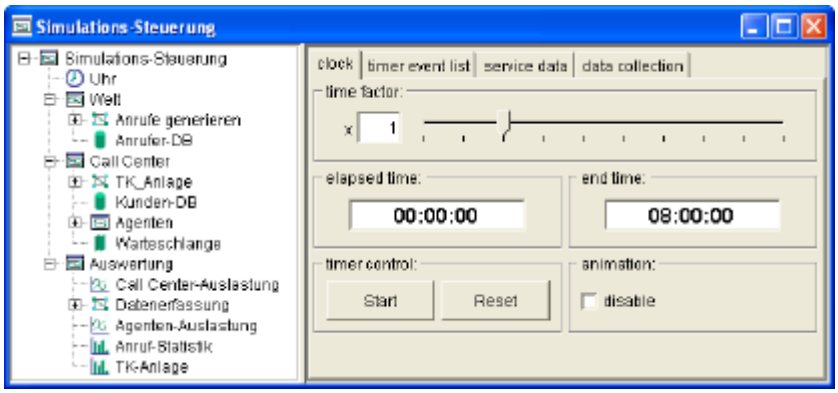


Abbildung 13 Fenster Simulations-Steuerung

(i) Simulationsfenster Welt

Das Fenster Welt (siehe Abbildung 14) besteht aus dem Simulations-Netzwerk Anrufe generieren sowie dem Datenbank-Bereich Anrufer-DB. Aus dieser Datenquelle werden die benötigten Daten für zu generierende Anrufe ermittelt, die sich außer persönlichen Daten aus folgenden Informationen zusammensetzen:

- Kundennummer;
- Name eines Wunschagenten (des bearbeitenden Agenten beim letzten CC-Anruf);
- zufällig erzeugte Wartezeit (die der Anrufer gewillt ist in der Warteschlange zu verweilen; zwischen ein und zehn Minuten);
- persönliche Erfahrung des Kunden;
- Grund des Anrufs.

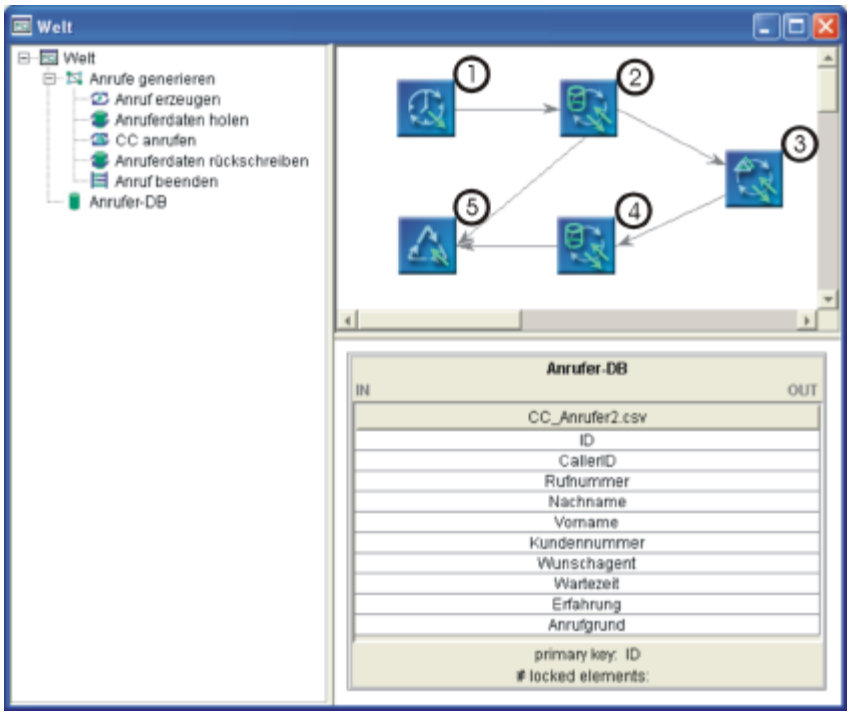


Abbildung 14 Simulationsfenster Welt

Dabei ist das Netzwerk Anrufe generieren so konfiguriert, dass sich hier folgender Ablauf ergibt: Die *Source*(1) erzeugt gemäß des zugehörigen Benutzercodes in unregelmäßigen Zeitintervallen leere Datenobjekte und leitet sie über den einzigen ausgehenden Konnektor an die *Datenstation*(2) weiter. Diese generiert daraus Anruf-Objekte, indem den Datenobjekten Informationen – beispielsweise Name, Telefonnummer und Anrufgrund – aus der Datenbank Anrufer-DB hinzugefügt werden. Der hier vorliegende Benutzercode (siehe Abbildung 15) führt anhand eines Anrufzählers eine Datenbank-Abfrage durch (siehe Zeile 05). Existiert kein dem Anrufzähler entsprechender Eintrag, wird der interne Zähler zurückgesetzt und eine erneute Abfrage durchgeführt (siehe Zeilen 24-26). Ist hingegen das betreffende Datenelement nicht verfügbar (siehe Zeilen 15-23), wird das leere Datenobjekt an *Datensenke*(5) weitergereicht und dort vernichtet.

```
01  //this = Referenz auf vorliegendes Usercode-Objekt
02  boolean fertig = false;
03  int alt_anrufZaehler = this.anrufZaehler;
04  while (!fertig) {
05    switch (table.existData("ID", this.anrufZaehler)) {
06      case SL_Table.ELEM_AVAIL : {
07             this.me = table.selectData(this.me, "ID", this.anrufZaehler);
08             int wartezeit = (int) (Math.random() * 10) + 1;
09             this.me.putString("Wartezeit", Integer.toString(wartezeit));
10             this.sendME("weiter", this.me);
11             this.anrufZaehler++;
12             fertig = true;
13             break;
14           }
15      case SL_Table.ELEM_LOCKED : {
16             this.anrufZaehler++;
17             this.sleep(100);
18             if (this.anrufZaehler == alt_anrufZaehler) {
19               this.sendME("fehler", this.me);
20               fertig = true;
21             }
22             break;
23           }
24      case SL_Table.ELEM_UNAVAIL : {
25             this.anrufZaehler = 0;
26           }
27    }
28  }
```

Abbildung 15 Benutzercode von Datenstation(2)

Liefert hingegen die Datenbank-Abfrage einen gültigen Datensatz, werden die gefundenen Anrufer-Informationen in das leere Datenobjekt eingetragen und an den Ausgang „weiter“ zum nachfolgenden *Consumer*(3) weitergeleitet, welcher anschließend die erzeugten Anruf-Objekte an den korrespondierenden *Provider* im Netzwerk TK-Anlage übergibt (was einem Anruf im Call Center entspricht). Nach erfolgter Abarbeitung im Call Center wird das Anrufer-Ojekt abschließend an *Consumer*(3) zurückgeleitet und an die nachfolgende *Datenstation*(4) übergeben. Hier werden die durch den Simulationslauf modifizierten Daten in die Anrufer-DB zurückgeschrieben und durch *Datensenke*(5) das nicht mehr benötigte Datenobjekt entfernt.

(ii) Simulationsfenster Call Center

Das Fenster Call Center (siehe Abbildung 16) besteht aus einer Datenkomponente Kunden-DB, einer Warteschlange für eingehende Anrufe, dem Netzwerk TK-Anlage sowie dem ausgelagerten Netzwerk-Fenster Agenten.

Die Warteschlange wird benötigt, um eingehende Call Center-Anrufe zu puffern. In regelmäßigen Abständen werden die eingereihten Anruf-Objekte der Warteschlange entnommen und getestet, ob ein freier Agent zur Bearbeitung bereit steht oder die maximale Verweildauer bereits überschritten ist. Anderenfalls werden die Anrufe wieder eingereiht.

Die Kunden-DB speichert Onlineakten, geordnet nach einer internen Integer-ID (Primärschlüssel). Weitere Bestandteile von Onlineakten sind CallerID, Rufnummer sowie persönliche Daten eines Kunden. Weiterhin werden Kundennummer (vergeben bei dem ersten Anruf des Kunden im Call Center), Name des letzten Agenten sowie Art des letzten Vorgangs gespeichert. Alle diese Werte sind String-Zeichenketten.

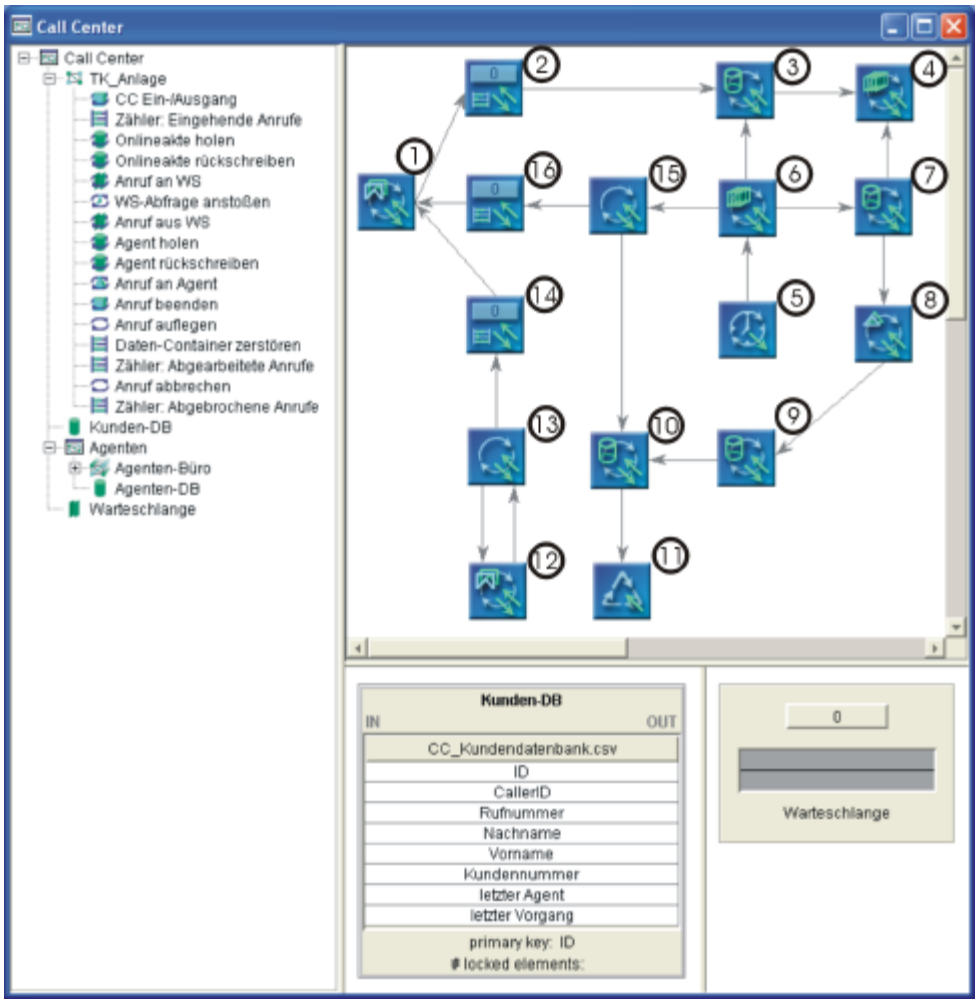


Abbildung 16 Simulationsfenster Call Center

Das Simulations-Netzwerk TK-Anlage nimmt eingehende Anrufe durch *Provider*(1) entgegen und erzeugt einen leeren Daten-Container, dem das empfangene Anrufer-Objekt hinzugefügt wird. Der Container wird samt Anruf an *Counter*(2) weitergeleitet, wo alle eingehenden Anrufe gezählt werden. Die nachfolgende *Datenstation*(3) übernimmt gemäß zugehörendem Benutzercode (siehe Abbildung 17) das Ermitteln der zu dem Anrufer gehörenden Onlineakte, die ebenfalls im Daten-Container abgelegt wird (siehe Zeilen 19-22). Sollte keine Onlineakte für den aktuellen Anrufer verfügbar sein, wird eine neue erzeugt (siehe Zeilen 10-15). Zudem wird aus dem vorliegenden Anruf-Objekt die Verweildauer bis zum Auflegen ausgelesen und anhand der aktuellen Simulationszeit ein 'Auflegezeitpunkt' errechnet (siehe Zeilen 05-06). Dieser Zeitstempel wird im Daten-Container gespeichert und zu einem späteren Zeitpunkt ausgelesen, um das Auflegen eines in der Warteschlange verweilenden Anrufers zu simulieren. Danach wird der Daten-Container mit Anrufer-Objekt und zugehöriger Onlineakte durch *Datenstation*(4) in die Warteschlange eingereiht und im Daten-Container der Parameter 'Akte gesperrt' initialisiert.

```
01  //this = Referenz auf vorliegendes Usercode-Objekt
02  this.me.putBool("Akte gesperrt", false);
03  SL_MoveableEntity anruf = this.me.getME("Anruf");
04  //Zeitstempel + Wartezeit für Einreihen in WS dem Daten-Container hinzufügen
05  int wartezeit = Integer.parseInt(anruf.getString("Wartezeit"));
06  this.me.putInt("Auflegezeitpunkt", (wartezeit * 60) + this.getSimulationTime());
07  String callerID = anruf.getString("CallerID");
08  SL_MoveableEntity onlineAkte =
        this.table.initBlankData((SL_MoveableEntity)SL_MoveableEntity.getInstance());
09  int res = this.table.existData("CallerID", callerID);
10  if (res == SL_Table.ELEM_UNAVAIL) {
11    onlineAkte.putString("CallerID", callerID);
12    onlineAkte.putString("letzter Agent", "-2");
13    this.table.insertData(onlineAkte, "CallerID", callerID);
14    res = SL_Table.ELEM_AVAIL;
15  }
16  if (res == SL_Table.ELEM_LOCKED) {
17    this.me.putBool("Akte gesperrt", true);
18  }
19  if (res == SL_Table.ELEM_AVAIL) {
20    onlineAkte = this.table.selectData(onlineAkte, "CallerID", callerID);
21    this.me.putME("Onlineakte", onlineAkte);
22  }
23  this.sendME("def", this.me);
```

Abbildung 17 Benutzercode von Datenstation(3)

Das Entnehmen von eingereihten Daten-Containern aus der Warteschlange wird durch *Source*(5) eingeleitet. Diese erzeugt in regelmäßigen Abständen (wenigen Sekunden) leere Container-Objekte und stößt somit die Abarbeitung des Benutzercodes von *Datenstation*(6) an (siehe Abbildung 18). Hier werden Daten-Container der Warteschlange entnommen (siehe Zeile 02) und der zuvor eingetragene 'Auflegezeitpunkt' mit der aktuellen Simulationszeit verglichen. Ist der Auflegezeitpunkt bereits verstrichen, wird der Daten-Container samt Anruf-Objekt und Onlineakte an *Station*(15) weitergereicht (siehe Zeilen 12-14). Diese verteilt die enthaltenen Daten-Objekte, sodass die Onlineakte zurückgeschrieben und das Anrufer-Objekt an das Simulationsfenster Welt zurückgegeben wird. *Counter*(16) übernimmt dabei das Zählen der abgebrochenen Anrufe.

```
01  //this = Referenz auf vorliegendes Usercode-Objekt
02  this.me = this.queue.dequeue();
03  if (this.me != null) {
04    if (this.me.getInt("Auflegezeitpunkt") > this.getSimulationTime()) {
05      if (this.me.getBool("Akte gesperrt")) {
06        this.sendME("Akte", this.me);
07      }
08      else {
09        this.sendME("Agent", this.me);
10      }
11    }
12    else {
13      this.sendME("auflegen", this.me);
14    }
15  }
```

Abbildung 18 Benutzercode von Datenstation(6)

Ist der Auflegezeitpunkt noch nicht erreicht, wird der beim Einreihen eingefügte Parameter 'Akte gesperrt' ausgelesen. Ist die Onlineakte nicht verfügbar (siehe Zeilen 05-07), wird das Datenobjekt an *Datenstation*(3) weitergeleitet, wo eine erneute Abfrage nach dem Status der zugehörigen Onlineakte durchgeführt wird. Im anderen Fall (Akte nicht gesperrt) wird das Datenobjekt an *Datenstation*(7) weitergereicht (siehe Zeilen 08-10). Diese Datenstation greift auf die Agenten-DB (im Fenster Agenten) zu und ermittelt anhand der vorliegenden Informationen im Anrufer-Objekt einen freien Agenten. Sind momentan alle Agenten beschäftigt, wird das Datenobjekt wieder über *Datenstation*(4) in die Warteschlange eingereiht. Anderenfalls wird der verfügbare Agent in dem Daten-Container nebst Anruf-Objekt und Onlineakte hinzugefügt und an *Consumer*(8) weitergereicht. Dieser übergibt das Datenobjekt an eines der Agenten-Netzwerke, wo die weitere Verarbeitung stattfindet. Nach

erfolgreicher Abarbeitung und Beenden des Anrufs wird das Datenobjekt an *Consumer*(8) zurückgereicht und über *Datenstation*(9) die aktuellen Daten des Agenten in die Agenten-Datenbank zurückgeschrieben. *Datenstation*(10) übernimmt das Rückschreiben der aktuellen Onlineakte in die Kunden-DB und über *Datensenke*(11) wird der nun leere Daten-Container entfernt.

Provider(12) nimmt Daten-Container von Agenten entgegen und leitet diese an *Station*(13) weiter. Hier wird das Auflegen des Anrufers nach einem Agentengespräch simuliert. Dazu wird das Anruf-Objekt aus dem Daten-Container entfernt und über *Counter*(14) an *Provider*(1) weitergereicht. Dieser übergibt das Anrufer-Objekt wiederum an den korrespondierenden *Consumer* im Netzwerk "Welt". Somit ist der Anrufer-Kreislauf geschlossen. Der noch vorhandene Daten-Container samt Onlineakte und Agent wird über *Provider*(12) wieder dem Agentennetzwerk zur weiteren Abarbeitung übergeben.

(iii) Simulationsfenster Agenten

Das ausgelagerte Fenster Agenten (siehe Abbildung 19) besteht aus der Datenbank-Komponente Agenten-DB und dem Multi-Netframe Agenten-Büro. Das Erzeugen der Agenten-Netzwerk-Kopien geschieht automatisch beim Starten der Simulation. Lediglich die maximale Anzahl der zu erzeugenden Kopien muss seitens des Benutzers zuvor festgelegt werden.

Die Datenkomponente Agenten-DB enthält Datensätze, die die einzelnen Agenten des Call Centers charakterisieren. Neben den persönlichen Daten werden Arbeitsbeginn und –zeit eines jeden „Mitarbeiters“ festgehalten. Weiterhin bezeichnet die 'aktive Zeit' die tatsächliche Arbeitsdauer eines Agenten. Die 'bearbeiteten Anrufe' speichern die abgearbeiteten Anrufe eines Agenten. 'Bildung' ist ein Parameter, anhand dessen der Bildungsgrad eines Agenten festgelegt wird. Dieser kann Einfluss auf die Bearbeitungsdauer eines Anrufers haben, wird in diesem Call Center-Modell jedoch nicht ausgewertet.

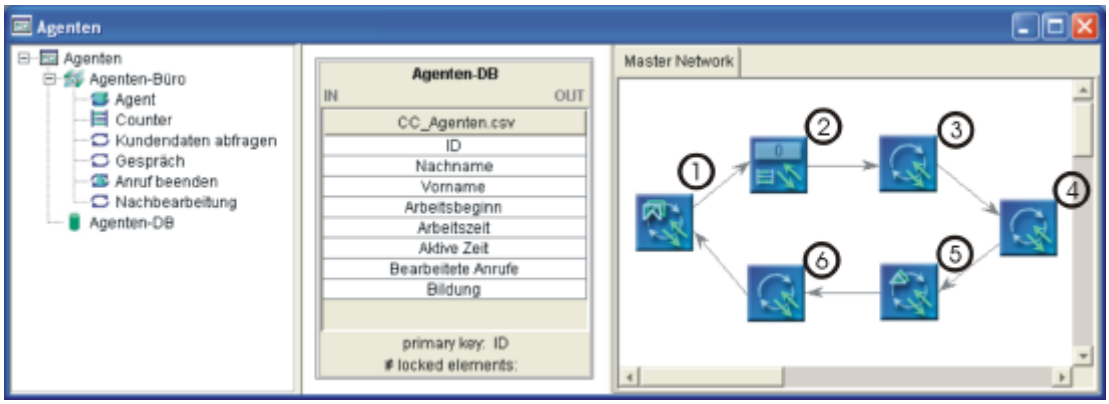


Abbildung 19 Simulationsfenster Agenten

Das Multi-Netzwerk Agenten-Büro nimmt Daten-Container (mit Anrufobjekt, zugehöriger Onlineakte und Agenteninformation) über *Provider*(1) entgegen und leitet diese an den nachfolgenden *Counter*(2) weiter, welcher die eingehenden Anrufer pro Agent registriert. *Station*(3) liest die vorhandenen Daten aus und füllt die Onlineakte mit Daten, falls es der erste Anruf im Call Center ist (Einfügen von Kundennummer, Name, Vorname, usw.). *Station*(4) simuliert das Führen eines Gesprächs, wobei die Gesprächsdauer per Zufall (zwischen 30 und 150 Sekunden) gesetzt wird. *Consumer*(5) übergibt den Daten-Container samt beinhalteten Daten-Objekten an den korrespondierenden *Provider* im Netzwerk TK-Anlage, wo das Auflegen simuliert wird. Der zurückkommende Daten-Container mit Onlineakte und Agenteninformation wird an *Station*(6) weitergereicht, welche die Nachbearbeitungsphase simuliert. Die Dauer der Nachbearbeitung wird hier ebenfalls per Zufall erzeugt (zwischen 30 und 60 Sekunden). Über *Provider*(1) wird das Datenobjekt wieder an die TK-Anlage zurückgegeben. Damit ist die Arbeit des Agenten beendet, und er wird von der TK-Anlage wieder als verfügbar markiert.

(iv) Simulationfenster Auswertung

Weiterer Bestandteil des zu simulierenden Call Center Modells ist die Auswertung (siehe Abbildung 20), die in diesem Beispiel durch ein Datenerfassungs-Netzwerk sowie zwei Plotter- und zwei Chart-Diagramme realisiert wurde. Im Netzwerk erzeugt *Source*(1) in regelmäßigen Abständen (15 Sekunden) Daten-Container und leitet sie an *Datenstation*(2) weiter. Aufgrund des vorliegenden Benutzercodes wird die Agenten-Datenbank durchlaufen

und die Gesamtauslastung der Agenten anhand der zur Zeit gesperrten und freien Agenten-Objekte ermittelt. Dieser Wert wird in den Daten-Container eingefügt und über *Variable*(3) von Plotter-Diagramm(I) graphisch dargestellt. Abschließend übernimmt die *Datensenke*(4) das Vernichten der Daten-Container.

Das Plotter-Diagramm(II) stellt die Call Center-Auslastung dar. Die rote Kurve zeigt die Anrufe im System, die blaue die Anrufe in der Warteschlange und die grüne die Differenz zwischen beiden (die zur Zeit in Bearbeitung befindlichen Anrufe).

Das Balken-Diagramm(III) repräsentiert die Anruf-Statistik, wobei je ein Balkenelement für eingehende Anrufe, bearbeitete Anrufe und abgebrochene Anrufe vorhanden ist.

Das Balken-Diagramm(IV) zeigt die Anzahl der eingereihten Anrufe in der Warteschlange.

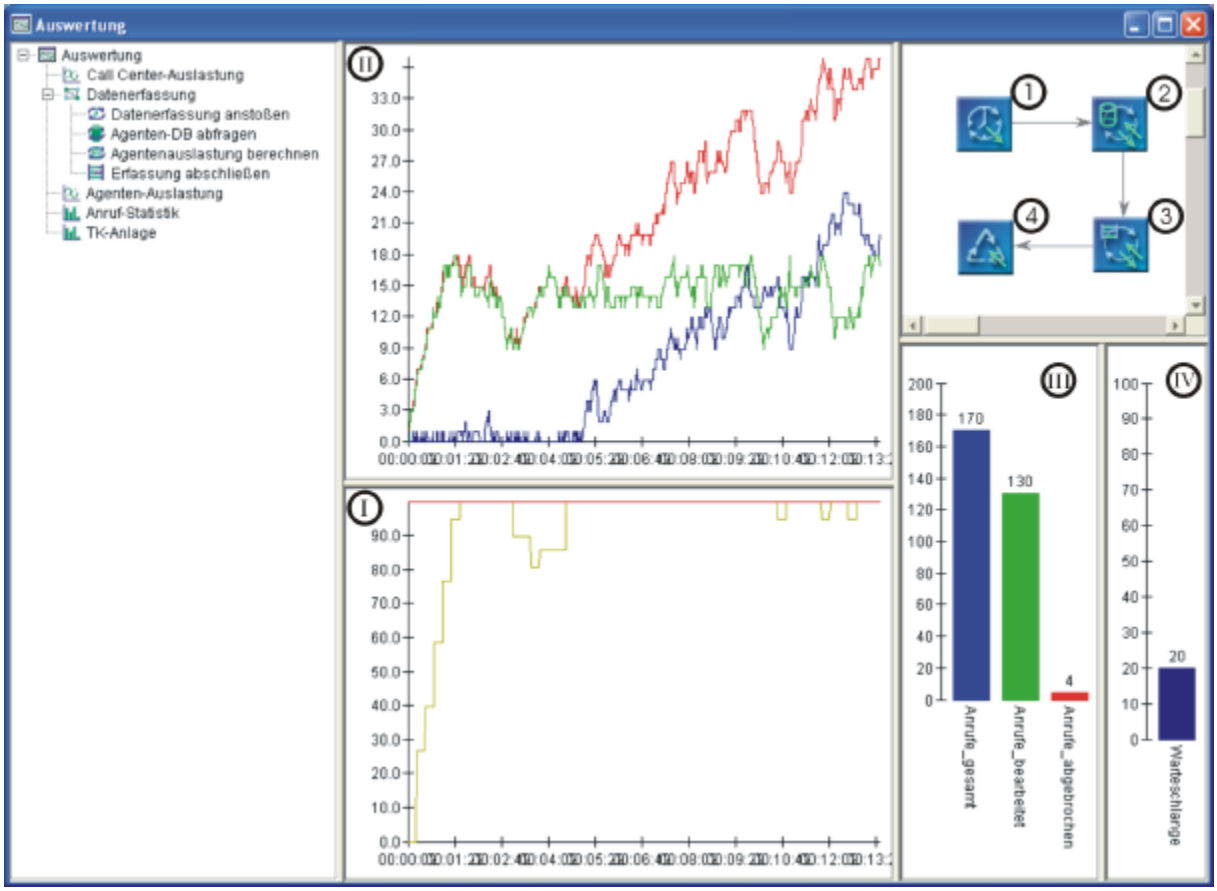


Abbildung 20 Simulationsfenster Auswertung

3.2 Auswertung

Das vorliegende Kapitel beschäftigt sich mit der Auswertung verschiedener Simulationsläufe, die mit der EnSiCC-Umgebung erstellt wurden. Grundlage ist das in Kapitel 3.1 vorgestellte Call Center Modell. Um anschauliche Simulationsergebnisse zu erhalten, wurden die Parameter „Anrufer-Ankunftsrate“ und „Anzahl der zur Verfügung stehenden Agenten“ in den verschiedenen Simulationsläufen variiert. Weiterhin wurde die Simulationszeit auf vier Stunden begrenzt. Die Simulationsanimation war abgeschaltet und 16fache Simulationsgeschwindigkeit eingestellt, sodass die Prozessoren der Rechensystemen nicht zu 100 Prozent ausgelastet und dadurch verfälschte Simulationsergebnisse ausgeschlossen werden konnten.

Im folgenden Verlauf werden die einzelnen Simulationsläufe und die Parameter-Einstellungen vorgestellt. Eine ausführliche Erläuterung der von der EnSiCC-Umgebung erzeugten Diagramme erfolgt in der Beschreibung der ersten Simulationskonfiguration. Die restlichen Simulationsläufe enthalten lediglich eine Kurzbeschreibung sowie die für den jeweiligen Simulationslauf aussagekräftigsten Diagramme.

Jeder Simulationslauf wurde auf drei verschiedenen Rechensystemen durchgeführt, um die Konsistenz der Daten und Diagramme zu untermauern. Alle Rechner verfügen über Java-Version 1.3.0-C, wobei Rechner 1 und 2 die Simulationsläufe unter Borland JBuilder 4 absolvierten und Rechner 3 die EnSiCC-Simulationsumgebung per Batch-Datei ausführte, um den Overhead der JBuilder-Entwicklungsumgebung zu unterdrücken (was jedoch keine Auswirkung auf den Ablauf der Simulationen hatte). Die Konfiguration der benutzten Rechensysteme enthält Tabelle 3.

Rechensystem	Rechnerkonfiguration
Rechner 1	AMD Athlon 1000 MHz, 512 MByte RAM, Windows XP Professional, EnSiCC per Borland JBuilder 4.
Rechner 2	AMD Athlon 1000 MHz Mobile, 256 MByte RAM, Windows XP Home, EnSiCC per Borland JBuilder 4.
Rechner 3	Intel Pentium III 1000 MHz Mobile, 256 MByte RAM, Windows XP Home, EnSiCC per Batch-Datei

Tabelle 3 Rechnerkonfigurationen

1. Simulationskonfiguration

In der ersten Simulationskonfiguration liegt eine konstante Ankunftsrate vor. Anrufer-Objekte erreichen das Call Center alle 5 Sekunden, d.h. 12 Anrufer pro Minute werden erzeugt. Die Anzahl der verfügbaren Call Center Agenten wurde variabel gestaltet, wobei Tabelle 4 die aktiven Agenten zu verschiedenen Simulationszeiten beinhaltet.

Simulationszeit	Anzahl aktiver Agenten
0:00:00	12
0:30:00	14
1:00:00	18
2:00:00	22
3:00:00	26

Tabelle 4 Aktive Agenten während Simulationslauf

Tabelle 5 liefert eine Übersicht aller verwendeter Simulationsparameter. In dieser ersten Simulationskonfiguration werden sie konstant gehalten.

Simulations-Parameter	Wert
Anrufer-Ankunftsrate	12 pro Minute
Verweildauer in Warteschlange	5 Minuten
Gesprächsdauer	90 Sekunden
Nachbearbeitungsdauer	45 Sekunden

Tabelle 5 Parameter der ersten Simulationskonfiguration

Folgendes Plotter-Diagramm (siehe Abbildung 21) stellt die Auslastung des Call Centers dar. Die rote Kurve zeigt die im Call Center befindlichen Anrufer, die blaue Kurve die Anrufe in der Warteschlange und die grüne die Anzahl der Agenten, die momentan mit der Bearbeitung eines Anrufs beschäftigt sind.

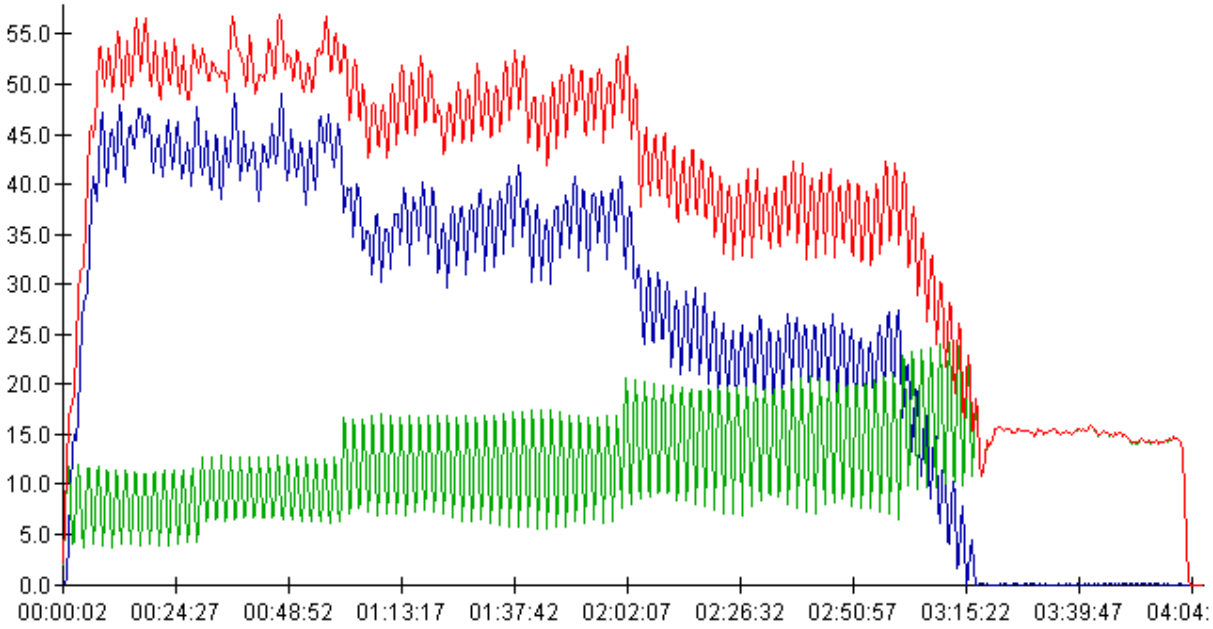


Abbildung 21 Call Center-Auslastung und Anzahl aktiver Agenten

Wie die grüne Kurve erkennen lässt, wurde die Zahl der aktiven Agenten während des durchgeführten Simulationslaufs ständig erhöht (vgl. Tabelle 4). Entsprechend der steigenden Anzahl zur Verfügung stehender Agenten nehmen die im Call Center (rote Kurve) beziehungsweise in der Warteschlange befindlichen Anrufer (blaue Kurve) ab. Nach etwa drei Stunden sind genügend (26) Agenten vorhanden, um die in der Warteschlange befindlichen Anrufer abzuarbeiten. Nachdem die Warteschlange vollkommen abgebaut wurde, werden

eingehende Anrufe sofort von den zur Verfügung stehenden Agenten bearbeitet und somit keine Anrufe mehr eingereiht (Verlauf der blauen Kurve auf der Diagramm-Abszisse). Nach Ablauf der Simulation (4 Stunden) werden keine Anrufe mehr entgegengenommen und die restlichen im System befindlichen Anrufer-Objekte von den Agenten abgehandelt.

Das folgende Plotter-Diagramm (siehe Abbildung 22) stellt die Auslastung der Agenten in Prozent dar (gelbe Kurve). Die rote Linie (100 Prozent) dient lediglich als Referenz.

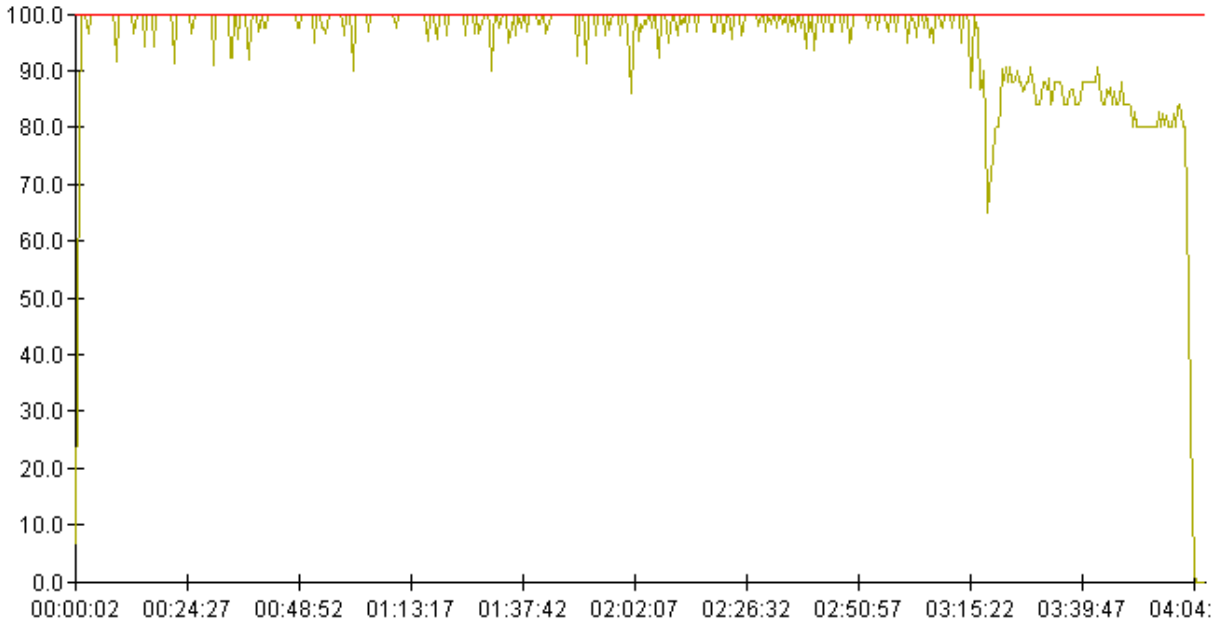


Abbildung 22 Agenten-Auslastung

Wie anhand der Auslastungskurve zu erkennen ist, werden die bis zu einer Zeit von etwa 3:15:00 Stunden eingesetzten Agenten nahezu 100 Prozent ausgelastet. Danach sinkt die Auslastung auf ein Niveau von zirka 85 Prozent. Dies deckt sich mit der in Abbildung 21 dargestellten Graphik. Nachdem dort die Warteschlange komplett abgearbeitet wurde, sind genügend Agenten vorhanden, um eingehende Anrufe sofort zu bearbeiten. Zudem werden einige der zu diesem Zeitpunkt 26 vorhandenen Agenten nicht mehr für die Bearbeitung von Anrufen herangezogen.

Im folgenden Diagramm (siehe Abbildung 23) stellt die blaue Kurve die Anrufer-Ankunftsrate (Anrufer pro Minute) dar. Der rote Graph hingegen zeigt die Abbrecher pro Minute.

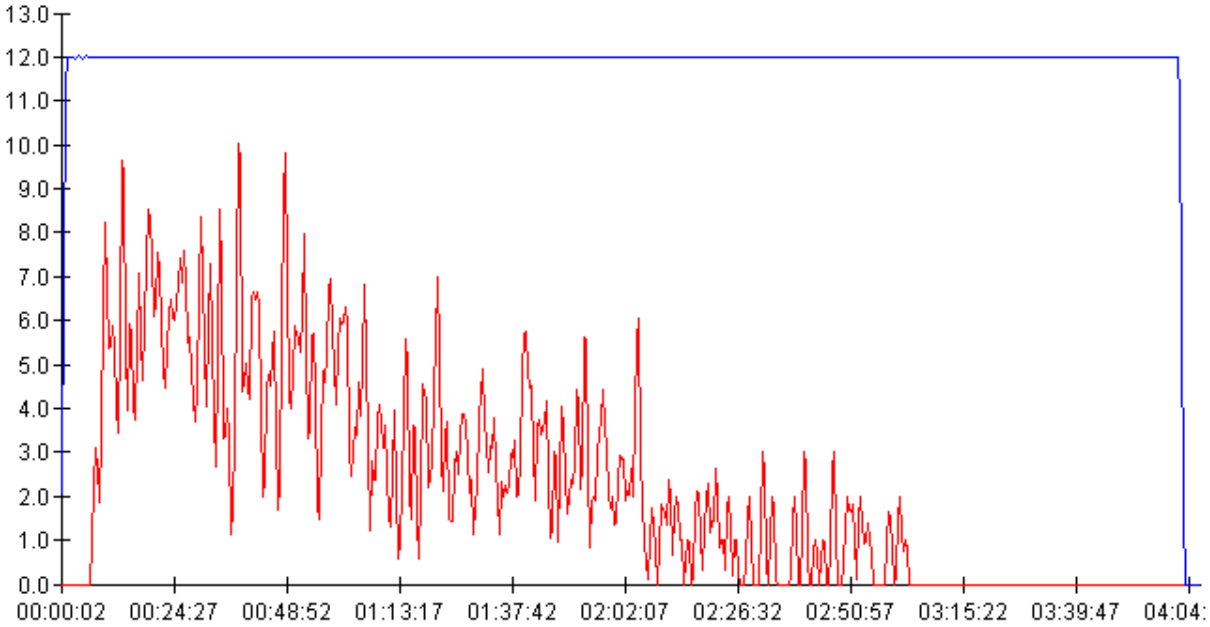


Abbildung 23 Ankunftsrate und Abbrecher pro Minute

Wie in Abbildung 23 zu erkennen ist, liegt eine konstante Ankunftsrate von 12 Anrufern pro Minute vor. Das entspricht der Voreinstellung des Simulationsmodells. Weiterhin zeigt die rote Kurve, dass nach einer Laufzeit von zirka 2:02:00 Stunden die Abbrecherquote auf einem niedrigeren Niveau verläuft als zuvor. Daraus lässt sich schlussfolgern, dass die zu diesem Zeitpunkt 22 aktiven Agenten die Anrufer-Ankunftsrate beinahe bewältigen können. Das erkennt man auch an dem niedrigen Niveau der blauen und roten Kurve aus Abbildung 21. Nach etwa 3:00:00 Stunden sind keine Abbrecher mehr zu erwarten, da nun genügend aktive Agenten vorhanden sind, um alle eingehende Anrufe zu bearbeiten. Das ist ebenfalls am Verlauf der blauen Kurve in Abbildung 21 zu erkennen, die ab diesem Zeitpunkt eine leere Warteschlange anzeigt.

Der Simulationslauf mit oben beschriebener Konfiguration lieferte folgendes Ergebnis:

	Rechner 1	**Rechner 2**	**Rechner 3**	**Mittelwert**
Anrufe gesamt	2.879	2.879	2.879	2.879
Anrufe bearbeitet	2.330	2.345	2.322	2.332
Anrufe abgebrochen	549	534	557	545

Tabelle 6 Ergebnis des Simulationslaufs

Alle Rechner erzeugten gleichwertige Ergebnisse, und die Kurvenverläufe der während des Simulationslaufs erzeugten Diagramme ergaben plausible Werte. Daraus wurde geschlussfolgert, dass sowohl Simulationssystem als auch -konfiguration zur Gewinnung stabiler Ergebnisse geeignet sind, sodass im nächsten Schritt Simulationsläufe mit zufallsgesteuerten Parametern durchgeführt werden konnten.

2. Simulationskonfiguration

Die zweite Simulationskonfiguration verfügt über eine zufällige Anrufer-Ankunftsrate. Im Mittel erreichen Anrufer das Call Center wie auch vorher mit einer Ankunftsrate von 12 Anrufern pro Minute. Die Anzahl zur Verfügung stehender Agenten wurde wie im ersten Simulationslauf variabel gestaltet (siehe Tabelle 4). Zusammenfassend zeigt Tabelle 7 alle Simulationsparameter.

Simulations-Parameter	Wert
Anrufer-Ankunftsrate	durchschnittlich 12 pro Minute
Verweildauer in Warteschlange	zufällig zwischen 1 und 10 Minuten
Gesprächsdauer	zufällig zwischen 30 und 150 Sekunden
Nachbearbeitungsdauer	zufällig zwischen 30 und 60 Sekunden

Tabelle 7 Parameter der zweiten Simulationskonfiguration

Für diese Konfiguration wurden drei Simulationsläufe durchgeführt, um die Zuverlässigkeit der Ergebnisse bei variablen Parametern zu überprüfen. Tabellen 8 bis 10 geben eine Übersicht der bei der Durchführung dieser Simulationsläufe erzielten Ergebnisse.

	Rechner 1	Rechner 2	Rechner 3	Mittelwert
Anrufe gesamt	2.928	2.858	2.778	2.855
Anrufe bearbeitet	2.321	2.320	2.340	2.327
Anrufe abgebrochen	607	538	438	528

Tabelle 8 Ergebnis 1. Simulationslauf

	Rechner 1	Rechner 2	Rechner 3	Mittelwert
Anrufe gesamt	2.915	2.926	2.835	2.892
Anrufe bearbeitet	2.283	2.328	2.336	2.316
Anrufe abgebrochen	632	598	499	576

Tabelle 9 Ergebnis 2. Simulationslauf

	Rechner 1	Rechner 2	Rechner 3	Mittelwert
Anrufe gesamt	2.902	2.894	2.968	2.921
Anrufe bearbeitet	2.316	2.356	2.348	2.340
Anrufe abgebrochen	586	538	620	581

Tabelle 10 Ergebnis 3. Simulationslauf

Abbildung 24 bis Abbildung 26 entstammen dem 2. Simulationslauf von Rechner 1. Trotz zufallsgesteuerter Parameter ist ein Kurvenverlauf ähnlich der Ergebnisse der ersten Simulationskonfiguration mit konstanten Parametern zu erkennen.

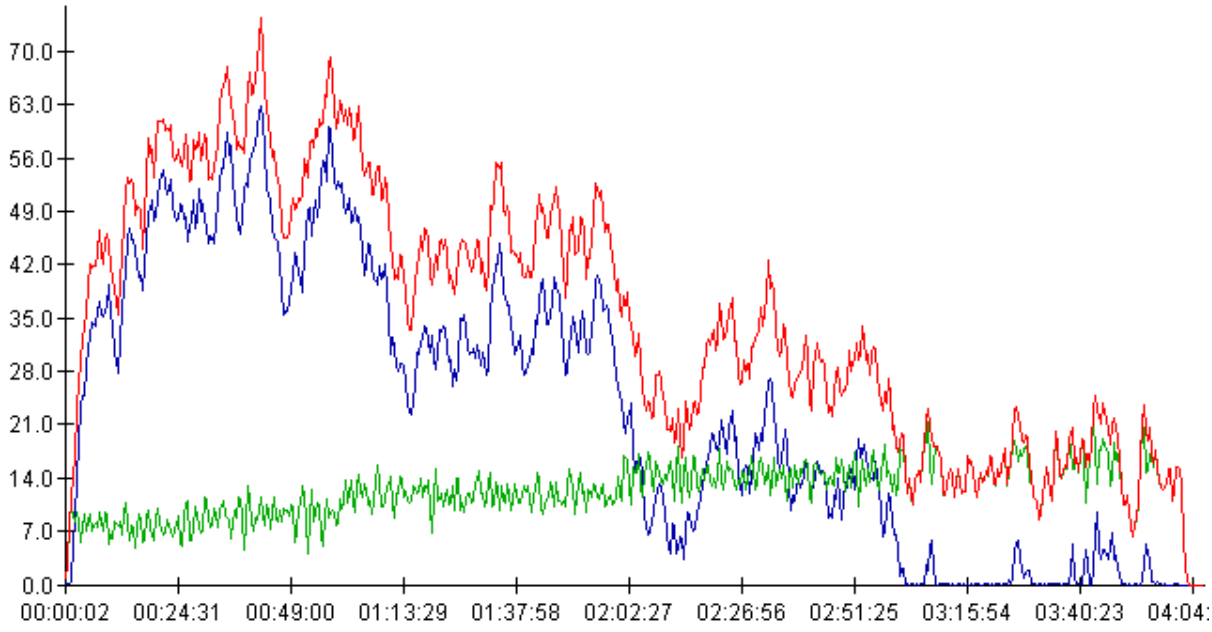


Abbildung 24 Call Center-Auslastung und Anzahl aktiver Agenten

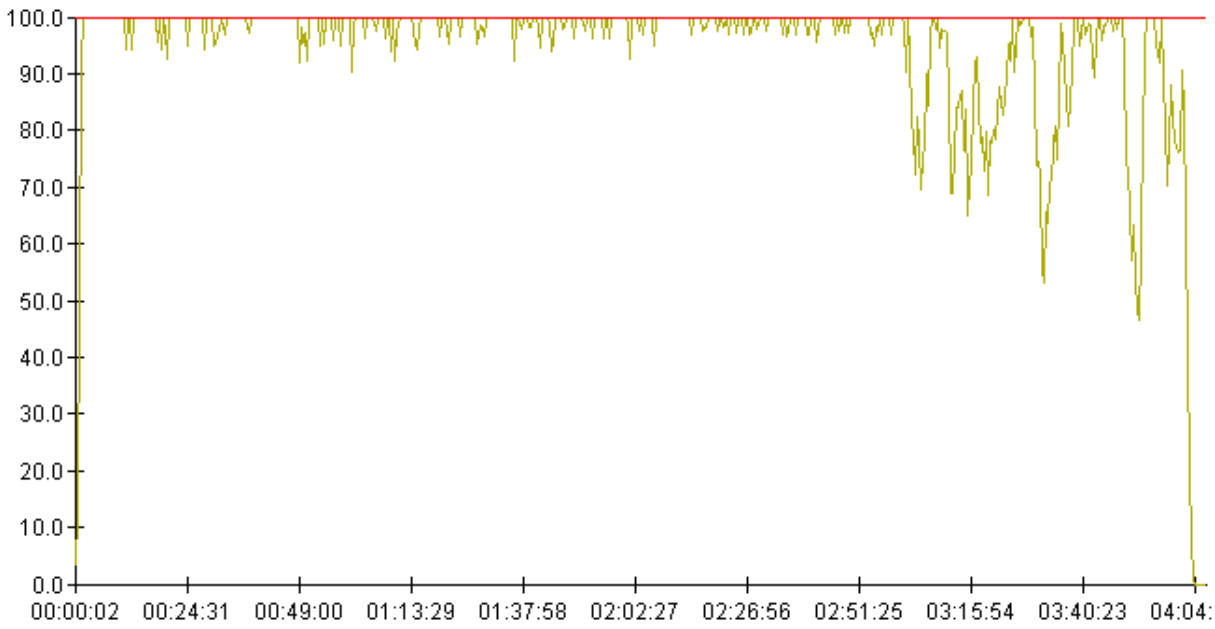


Abbildung 25 Agenten-Auslastung

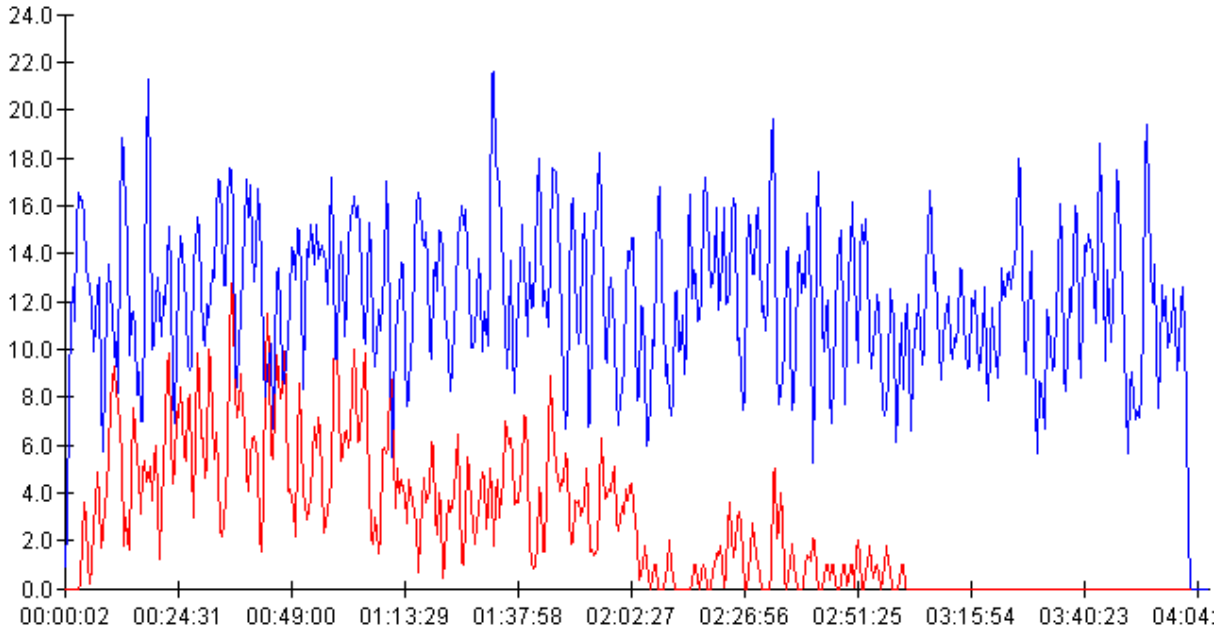
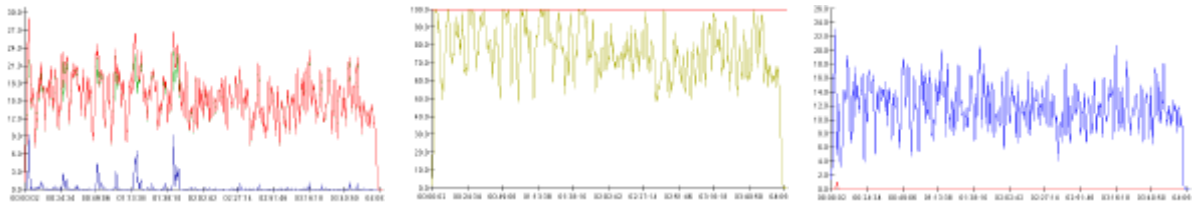
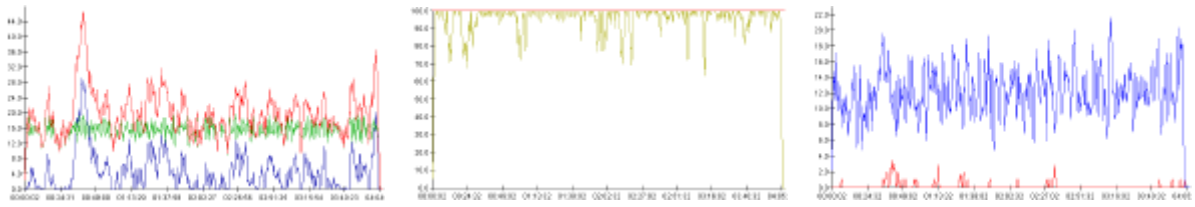


Abbildung 26 Ankunftsrate und Abbrecher pro Minute

3. Simulationskonfiguration

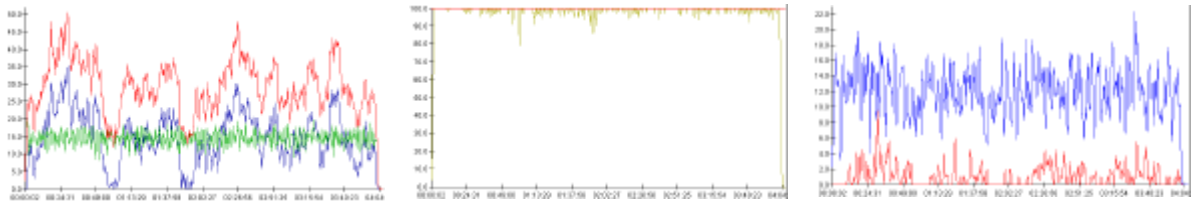
Die hier benutzten zufallsgesteuerten Parameter entsprechen denen der zweiten Simulationskonfiguration (siehe Tabelle 7). Lediglich die Anzahl der Agenten bleibt während eines Simulationslaufs konstant. Um eine optimale Agentenbesetzung für das Call Center-Modell zu ermitteln, wurden mehrere Läufe mit verschiedener Agentenzahl durchgeführt. Folgende Tabellen enthalten die von der Simulationsumgebung erzeugten Ergebnisse für die einzelnen Läufe:

Simulationslauf mit 30 Agenten				
	Rechner 1	**Rechner 2**	**Rechner 3**	**Mittelwert**
Anrufe gesamt	2.885	2.977	2.852	2.905
Anrufe bearbeitet	2.885	2.977	2.851	2.904
Anrufe abgebrochen	0	0	1	1
Diagramme von Rechner 3: 				

Simulationslauf mit 24 Agenten				
	Rechner 1	**Rechner 2**	**Rechner 3**	**Mittelwert**
Anrufe gesamt	2.855	2.923	2.877	2.885
Anrufe bearbeitet	2.777	2.882	2.782	2.814
Anrufe abgebrochen	78	41	95	71
Diagramme von Rechner 2: 				

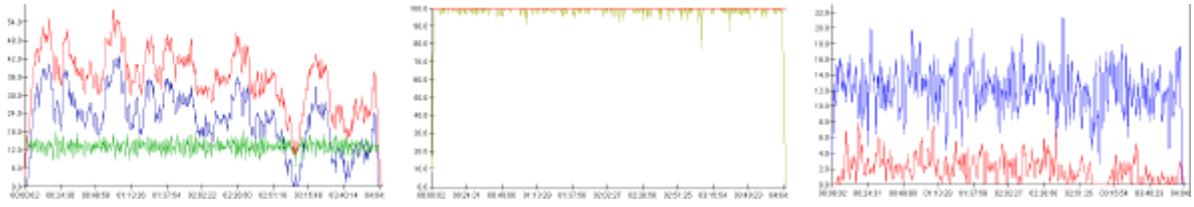
Simulationslauf mit 22 Agenten				
	Rechner 1	**Rechner 2**	**Rechner 3**	**Mittelwert**
Anrufe gesamt	2.882	2.988	2.936	2.935
Anrufe bearbeitet	2.609	2.674	2.575	2.619
Anrufe abgebrochen	273	314	361	316

Diagramme von Rechner 1:



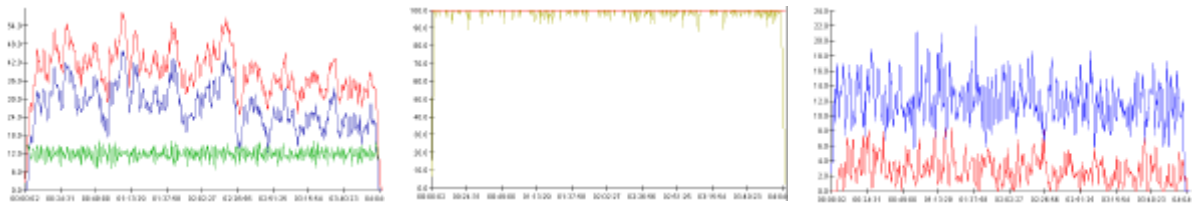
Simulationslauf mit 20 Agenten				
	Rechner 1	**Rechner 2**	**Rechner 3**	**Mittelwert**
Anrufe gesamt	2.875	2.940	2.886	2.900
Anrufe bearbeitet	2.401	2.464	2.452	2.439
Anrufe abgebrochen	474	476	434	461

Diagramme von Rechner 3:



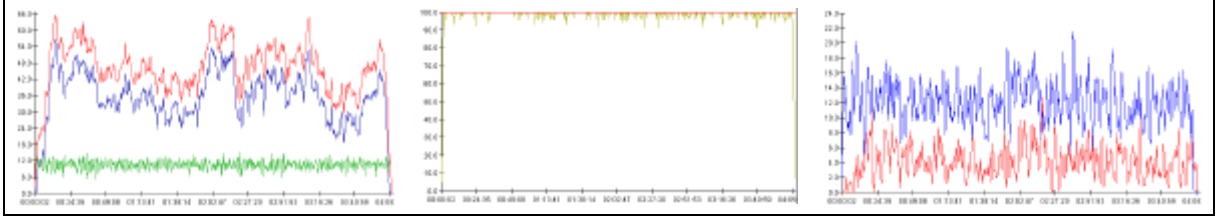
Simulationslauf mit 18 Agenten				
	Rechner 1	**Rechner 2**	**Rechner 3**	**Mittelwert**
Anrufe gesamt	2.933	2.868	2.830	2.877
Anrufe bearbeitet	2.206	2.188	2.137	2.177
Anrufe abgebrochen	727	680	693	700

Diagramme von Rechner 2:



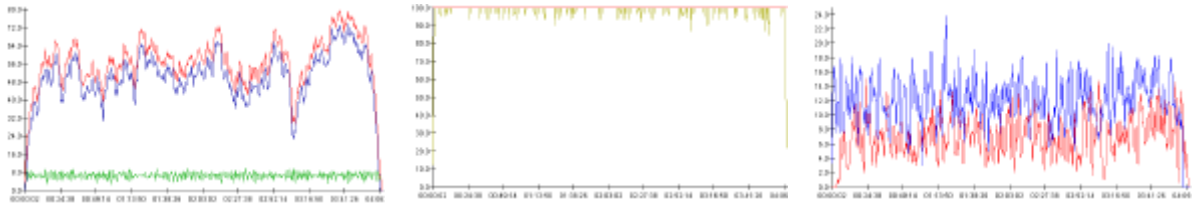
Simulationslauf mit 16 Agenten				
	Rechner 1	**Rechner 2**	**Rechner 3**	**Mittelwert**
Anrufe gesamt	2.938	2.837	2.942	2.906
Anrufe bearbeitet	1.922	1.921	2.000	1.948
Anrufe abgebrochen	1.016	916	942	958

Diagramme von Rechner 1:



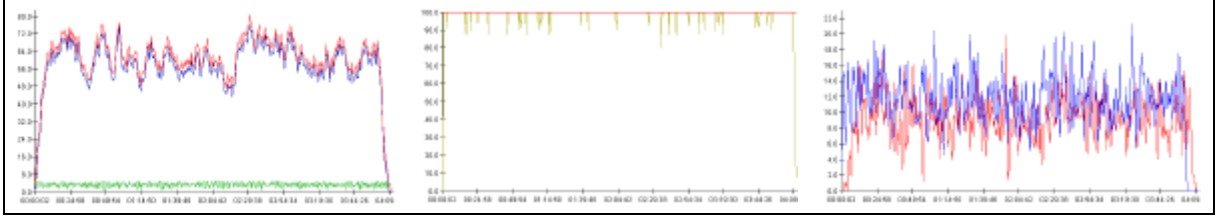
Simulationslauf mit 10 Agenten				
	Rechner 1	**Rechner 2**	**Rechner 3**	**Mittelwert**
Anrufe gesamt	2.938	2.942	2.999	2.960
Anrufe bearbeitet	1.196	1.238	1.220	1.218
Anrufe abgebrochen	1.742	1.704	1.779	1.742

Diagramme von Rechner 2:



Simulationslauf mit 5 Agenten				
	Rechner 1	**Rechner 2**	**Rechner 3**	**Mittelwert**
Anrufe gesamt	2.810	2.780	2.843	2.811
Anrufe bearbeitet	604	597	601	601
Anrufe abgebrochen	2.206	2.183	2.242	2.210

Diagramme von Rechner 3:



Zusammenfassend sind in Abbildung 27 alle Mittelwerte der für die dritte Simulationskonfiguration durchgeführten Läufe aufgetragen.

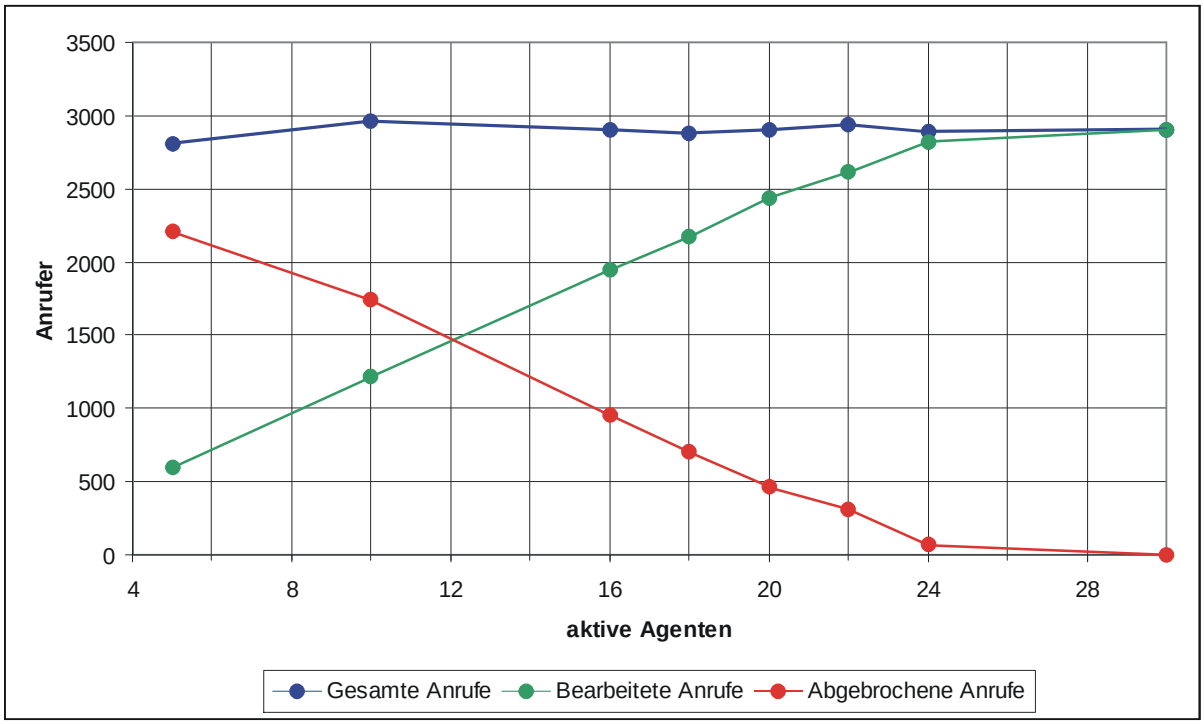


Abbildung 27 Mittelwerte der Ergebnisse

Als weiteres Bewertungskriterium zeigt Abbildung 28 die Auslastung der Agenten für die absolvierten Simulationsläufe. Als Datenquelle dienen die bei jeder Durchführung einer Simulation gespeicherten Zeiten, in denen ein Agent aktiv an der Bearbeitung eingehender Anrufe beteiligt war (vgl. Parameter 'aktive Zeit' in der Beschreibung des Agenten-Simulationsfensters in Kapitel 3.1).

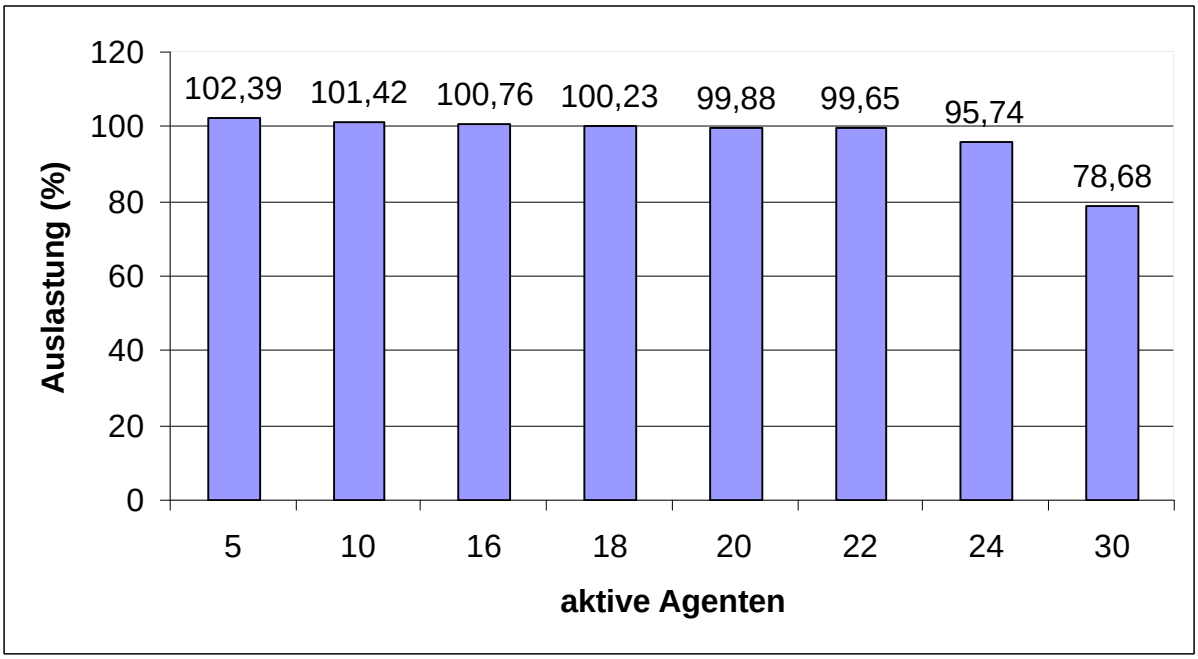


Abbildung 28 Agentenauslastung[1]

Betrachtet man die beiden vorhergehenden Diagramme, scheint eine Konfiguration mit 22 bis 24 aktiven Agenten den Ansprüchen eines wirtschaftlichen Call Centers zu entsprechen. Es wird eine hohe Anzahl eingehender Anrufe bearbeitet, die Abbruchquote ist gering und die Agenten sind nahezu vollkommen ausgelastet. Ab einer Anzahl von 24 aktiven Agenten ist keine Verbesserung der Leistung mehr zu erwarten, da sich die Abbrecherquote nur noch geringfügig verändert.

Wie die durchgeführten Simulationsläufe zeigen, ist es durchaus möglich, mit der entwickelten EnSiCC-Simulationsumgebung Ergebnisse für den Aufbau eines Call Centers zu erhalten. Durch Verändern anderer relevanter Parameter oder Hinzufügen weiterer Datenquellen sind detailliertere und aussagekräftigere Resultate zu erzielen.

[1] Eine Auslastung von über 100% bedeutet, dass die Agenten über die reguläre Arbeitszeit hinaus mit der Bearbeitung der Anrufe beschäftigt waren.

Literatur

AESOP GmbH (1998): „SIMPLE++ 5.0 – Referenzhandbuch."

Gilbert, N.; Troitzsch, K. G. (1999): „Simulation for the Social Scientist." Open University Press.

Kreutzer, W. (1986): „System Simulation: Programming Styles and Languages." Addison-Wesley, Sydney.

Niehl, G. (2000): „Call Center in der Simulation." Diplomarbeit an der Universität Koblenz-Landau, Abteilung Koblenz.

Poschmann, F.-P. (2000): „Call Center-Simulation." In: Call Center profi 8-9/2000, S.34-36.

Sun Microsystems, Inc. (2000): „Java™ 2 SDK Documentation, Version 1.3."

Ullenboom, C. (2001): „Java ist auch eine Insel." Galileo Press.