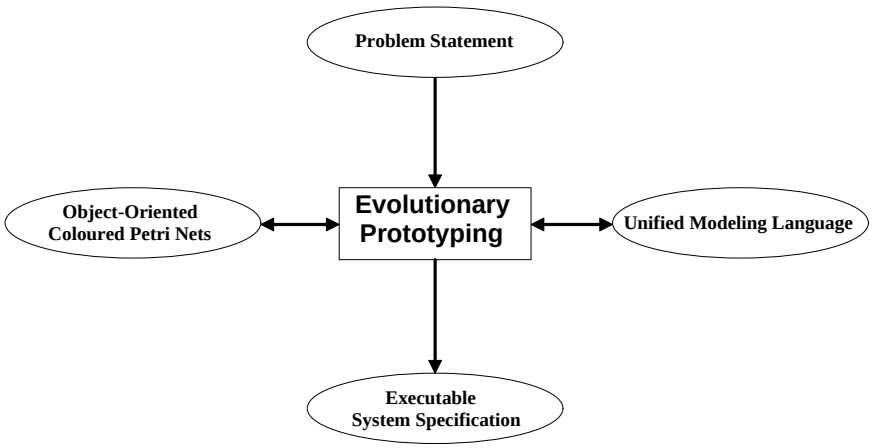
Untersuchung der Einsatzmöglichkeiten von Petrinetz-Konzepten in der objektorientierten Analyse am Fallbeispiel eines Reiseunternehmens



DIPLOMARBEIT

von

Marc Netzebandt

29. Januar 1999

Universität Hamburg

Fachbereich Informatik

Arbeitsbereich Theoretische Grundlagen der Informatik

Betreuer: Dr. Daniel Moldt

Zweitbetreuer: Prof. Dr. Florian Matthes

Überblick

Diese Arbeit beschäftigt sich mit der Systemanalyse aus dem Bereich der objektorientierten Softwareentwicklung. Das Ziel ist die Untersuchung der Einsatzmöglichkeiten von objektorientierten Petrinetzen zur Entwicklung einer ausführbaren Systemspezifikation. Eine Vorgehensweise zum Einsatz der Petrinetze wird im Rahmen dieser Arbeit entwickelt und an einem Fallbeispiel erprobt.

Zur Einarbeitung in den Bereich der objektorientierten Systemanalyse wird aus historischen Gründen die Object Modeling Technique (OMT) betrachtet und als aktuelle Modellierungssprache die standardisierte Unified Modeling Language (UML) vorgestellt. Mit diesen beiden Ansätzen werden Analysemodelle des Fallbeispiels erstellt und diskutiert. Da keine Techniken zur Entwicklung von ausführbaren Systemspezifikationen bereitgestellt werden, können keine Vorgehensweisen mit Prototyping eingesetzt und damit nicht dessen Vorteile genutzt werden.

Viele Ansätze haben gezeigt, daß sich Petrinetze zur Modellierung von dynamischen Systemaspekten eignen und ausgeführt werden können. Bei der prozeßorientierten, funktionalen Modellierung eines Reiseunternehmens mit gefärbten Petrinetzen in meiner Studienarbeit ergaben sich die folgenden Probleme:
Die modellierten Systemfunktionen sind stark ineinander verzahnt, wodurch sich Änderungen beim Prototyping nur schwierig vornehmen lassen. Die Funktionen haben keine standardisierte Schnittstelle und die Netze keine einheitliche Strukturierung. Dadurch können diese nicht problemlos wiederverwendet werden.

Als Lösungsansatz werden objektorientierte Konzepte auf Petrinetze übertragen. Umgesetzt werden die Konzepte der Klassen und Objekte, Assoziationen, Vererbung und Polymorphismus. Mechanismen zur Kommunikation zwischen Objekten werden vorgestellt und Konzepte zur verteilten Simulation der Petrinetzmodelle erläutert. Zur Erstellung der Petrinetze und zur Simulation wird das Werkzeug Design/CPN verwendet, das leider nicht die direkte Entwicklung von objektorientierten Petrinetzen unterstützt.

Die Entwicklung der ausführbaren Systemspezifikation des Reiseunternehmens beginnt mit der Erstellung eines Use-Case-Diagramms auf der Basis der textuellen Systembeschreibung. Dieses Diagramm bildet die Grundlage zur Entwicklung eines Klassendiagramms. Zu diesem werden Interaktionsdiagramme entworfen. Aus dem Klassendiagramm werden die objektorientierten Petrinetze derart abgeleitet, daß die Methodensignaturen und die Attribute als Netznotation das Gerüst für das evolutionäre Prototyping bilden. Die Interaktionsdiagramme unterstützen die Definition der Methoden und das Testen der modellierten Szenarien. In einem inkrementellen Prozeß werden, bis zur Fertigstellung der ausführbaren Systemspezifikation als objektorientiertes Petrinetzmodell, alle eingesetzten Techniken betrachtet.

Durch die objektorientierten Petrinetze werden die oben genannten Probleme aus der Studienarbeit teilweise gelöst. Änderungen können beim Prototyping durch die Entkopplung der Systemfunktionen einfacher vorgenommen werden und durch die einheitliche Schnittstelle der Methoden und die standardisierte Struktur der Netze wird die Wiederverwendbarkeit erhöht.
Die Klassen lassen sich gut aus den Klassendiagrammen ableiten. Die Realisierung der Assoziationen erfordert die Modellierung komplexer Protokolle. Für die Implementation der Methoden können Konstrukte zur Modellierung von Nebenläufigkeit, Sequentialität, synchronem bzw. asynchronem Methodenaufruf, Fallunterscheidungen, Schleifen und wechselseitigem Ausschluß verwendet werden. Dadurch wird eine gute Visualisierung der Algorithmen erreicht, die als Diskussionsgrundlage dient.

INHALT

1 EINLEITUNG ---------- 6
1.1 Systemanalyse ---------- 6
1.2 Petrinetze ---------- 7
1.3 Minotours: Historie zum Fallbeispiel ---------- 8
1.4 Rahmenbedingungen ---------- 9
1.5 Ziele und Aufbau ---------- 10

2 OBJEKTORIENTIERTE ANALYSETECHNIKEN ---------- 11
2.1 Object Modeling Technique ---------- 11
2.2 Unified Modeling Language ---------- 24

3 SPEZIFIKATION VON MINITOURS MIT OMT UND UML ---------- 37
3.1 Minitours ---------- 37
3.2 Spezifikation von Minitours mit OMT ---------- 38
3.3 Spezifikation von Minitours mit UML ---------- 64
3.4 Ergebnisse der objektorientierten Analyse ---------- 76

4 OBJEKTORIENTIERTE KONZEPTE FÜR PETRINETZE ---------- 77
4.1 Historie ---------- 77
4.2 Objekt- und Klassennetze ---------- 78
4.3 Nachrichtenformat und Methodenaufruf ---------- 87
4.4 Kommunikation ---------- 96
4.5 Assoziationen, Multiplizität und Aggregation ---------- 98
4.6 Vererbung und Polymorphismus ---------- 101
4.7 Beispiel: Listenklassen ---------- 104
4.8 Werkzeugunterstützung ---------- 111
4.9 Einbettung von OOCPN in die Analyse ---------- 119

5 EINSATZ VON OOCPN ZUR SPEZIFIKATION VON MINITOURS ---------- 125
5.1 Verwendete Petrinetz-Notation ---------- 125
5.2 Vorgehen ---------- 129
5.3 Modellierungsentscheidungen ---------- 140
5.4 Hierarchie des Petrinetzes ---------- 153
5.5 Ausführung der Spezifikation ---------- 155
5.6 Ergebnisse und Bewertung ---------- 157

6 ZUSAMMENFASSUNG UND AUSBLICK ---------- 161
6.1 Zusammenfassung ---------- 161
6.2 Ausblick ---------- 162

7 LITERATURANGABEN ---------- 163

8 ANHANG ---------- 167

VORWORT

An dieser Stelle möchte ich mich bei den Personen bedanken, die mich bei dieser Diplomarbeit unterstützt haben.

Daniel Moldt danke ich für die Betreuung dieser Arbeit. In vielen Gesprächen gab er mir wertvolle Hinweise zur Systemanalyse und bei der Entwicklung der Netze, sowie Tips zur Gestaltung der Arbeit.

Florian Matthes danke ich für die Zweitbetreuung der Arbeit und die spontanen Gesprächstermine zur Beratung im Bereich der objektorientierten Systementwicklung.

Frank Wienberg danke ich für die Realisierung der technischen Umsetzung der objektorientierten Petrinetze und die vielen Beratungen in technischen Fragen.

Adriana Engelhardt danke ich für die gemeinsame Einarbeitung in das Thema der objektorientierten Konzepte für Petrinetze und die Diskussion des Fallbeispiels.

Heiko Rölke und Wulf Harder danke ich für das Korrekturlesen dieser Arbeit und inhaltliche Diskussionen. Weiterer Dank für das Korrekturlesen geht an Aina und Vija.

Der Firma Syncrosoft danke ich für die Bereitstellung der Hard- und Softwarevoraussetzungen zur Verwendung von Rational Rose zur Modellierung des Fallbeispiels mit der Unified Modeling Language.

Meinen Eltern, Marianne und Uwe, danke ich dafür, daß sie mir dieses Studium ermöglicht haben.

1 EINLEITUNG

In den neunziger Jahren hat sich im Bereich der Softwareentwicklung die Objektorientierung durchgesetzt. Während in den achtziger Jahren die Objektorientierung nur den Bereich der Programmierung beeinflußte, findet man sie z.Zt. im gesamten Softwareentwicklungsprozeß, d.h. bei der Analyse, dem Design und der Implementation. Objektorientierte Systementwicklung erhöht die Wiederverwendbarkeit von Software. Aufgrund der Standardisierung der Unified Modeling Language (UML) als Notation zur objektorientierten Analyse und Design von Systemen, wird diese zunehmend zur Softwareentwicklung eingesetzt. Ein besonders wichtiges Gebiet ist die Systemanalyse, bei der beschrieben wird, was ein System leistet, nicht wie dieses implementiert wird. Die Use-Cases eignen sich zur Darstellung der Systemabgrenzung und mit den Klassendiagrammen können die statischen Aspekte von komplexen Systemen sehr gut modelliert werden. Andere Techniken stehen zur Visualisierung der dynamischen Aspekte zur Verfügung. Es handelt sich um Interaktionsdiagramme, Aktivitätsdiagramme und Zustandsdiagramme.

Speziell für die Modellierung der dynamischen Systemaspekte haben verschiedene Autoren vorgeschlagen Petrinetze einzusetzen. In Kapitel 4.1 wird auf einige Autoren und deren Ansätze verwiesen. Gründe für den Einsatz von Petrinetzen sind die Ausführbarkeit der Netze, die wenigen Grundbestandteile der Netze, adäquate Darstellung von Nebenläufigkeit und Ansätze zur Untersuchung des Systemverhaltens und von Systemeigenschaften aufgrund der mathematischen Definition der Netze. Durch die Kombination von Simulation und Prototyping können Fehlentwicklungen frühzeitig erkannt werden. Besonders die adäquate Modellierung von Nebenläufigkeit ist zu betonen, da Methoden bisher meistens aufgrund der verwendeten Notation unnötig sequentiell beschrieben werden.

In dieser Arbeit werden unterschiedliche Ansätze zur Umsetzung objektorientierter Konzepte für Petrinetze vorgestellt. Jeder Ansatz realisiert auf zum Teil unterschiedliche Art die aus der objektorientierten Programmierung bekannten Konzepte der Klassen, der Klassenvariablen und der statischen Methoden und der Objekte mit ihren Attributen und Methoden. Weitere Konzepte wie Vererbung, Interfaces, Assoziationen und Polymorphismus werden nicht bei allen Petrinetz-Ansätzen umgesetzt bzw. erfahren eine unterschiedliche Gewichtung.

Zur Entwicklung einer ausführbaren Systemspezifikation wird hier ein iteratives Vorgehen vorgeschlagen und am Fallbeispiel erprobt, bei dem einige Techniken der Unified Modeling Language und Petrinetze verwendet werden. Ausgehend von einem Use-Case-Diagramm, einem ersten Klassendiagramm und Interaktionsdiagrammen werden Petrinetz-Prototypen generiert und simuliert.

In diesem ersten Kapitel folgt zunächst eine kurze Einführung in die Themen Systemanalyse und Petrinetze, sowie die Betrachtung der prozeßorientierten Spezifikation eines Reiseunternehmens mit gefärbten Petrinetzen. Nach der Erläuterung einiger Rahmenbedingungen werden die Ziele und der Aufbau der Arbeit erläutert.

1.1 Systemanalyse

Das Ziel der Systemanalyse ist die Erfassung, Analyse und Modellierung von Anforderungen eines zu entwickelnden Softwaresystems ([Balzert 96]). Durch die zu Beginn der siebziger Jahre entwickelten strukturierten Programmiersprachen (u.a. [Dijkstra 72], [Wirth 72]) wurde der bis in die achtziger Jahre vorherrschende strukturierte Entwurf und die strukturierte Analyse (u.a. [DeMarco 79], [McMenamin und Palmer 88], [Hatley 87], [Yourdon 89]) geformt. Obwohl die Konzepte der Objektorientierung seit dem Anfang der achtziger Jahre bekannt sind (Smalltalk80, C++, Eiffel), gibt es

den Ansatz zum objektorientierten Entwurf (u.a [Meyer 88], [Wirfs-Brock et al. 90], [Booch 91] und [Coad/Yourdon 90]) erst seit Mitte/Ende der achtziger Jahre und zur objektorientierten Systemanalyse (OOA) erst seit dem Ende der achtziger Jahre und dem Anfang der neunziger Jahre ([Rumbaugh et al. 91], [Booch 94], u.a.). Seit dieser Zeit werden objektorientierte Programmiersprachen zunehmend eingesetzt, weil sich durch ihren Einsatz die Wiederverwendung bereits geschriebenen Codes erhöht und komplexe Systeme fehlerfreier entworfen werden können. Unterschiedliche Ansätze zur Verwendung der objektorientierten Konzepte in der Systemanalyse wurden von verschiedenen Autoren vorgeschlagen. Mitte der neunziger Jahre begannen G. Booch und J. Rumbaugh an einer Standardisierung der Techniken zu arbeiten, wodurch sie sich erhöhte Akzeptanz bzgl. des Einsatzes dieser Techniken seitens der Software-Entwickler erhofften. Unter Mitwirken von I. Jacobson ist das Ergebnis der "drei Amigos" seit Mitte 1997 die Unified Modeling Language. Bei dieser handelt es sich um einen Standard der Object Management Group (OMG). Information findet man unter den Internetadressen 'www.omg.org' und 'www.rational.com/uml/resources/documentation/'. Zur Version 1.1 können von der zuletzt angegebenen Adresse die folgenden Dokumente bezogen werden: [UML 1.1 Summary], [UML 1.1 Notation-Guide], [UML 1.1 Semantics], [UML 1.1 Extension for Objectory Process for Software Engineering], [UML 1.1 Extension for Business Modeling], [Object Constraint Language Specification]. Ein weiterer Grund für den vermehrten Einsatz von objektorientierter Systementwicklung ist die zunehmende Bedeutung des Internets und der damit verbundene Einsatz der plattformunabhängigen, objektorientierten Programmiersprache Java ([Cornell 97]).

Viele der z.Zt. verwendeten Vorgehensmodelle zur Softwareentwicklung sehen Iterationen über den gesamten Softwareentwicklungsprozeß, wie z.B. in [Quatrani 98] vor, und bei vielen steht die Kundenorientierung im Vordergrund. Gerade für diesen letztgenannten Aspekt eignet sich Prototyping und damit verbundenes, häufiges Kundenfeedback. Erstrebenswert sind also Techniken zur Erstellung von ausführbaren Prototypen. Eine Möglichkeit bietet der Einsatz von Petrinetzen. Entsprechend der o.g. historischen Entwicklung der Systemanalyse, wurden Petrinetze zunächst in Verbindung mit der strukturierten Analyse eingesetzt. Vorteile sind nicht nur die Ausführbarkeit der Prototypen, sondern auch die Möglichkeiten der Modellierung von Nebenläufigkeit. Außerdem gibt es eine mathematische Definition von Petrinetzen, die Mechanismen zur Untersuchung der modellierten Systeme bezüglich Verklemmungen und ähnlichen Fehlern zur Verfügung stellt. Im Zuge der weiten Verbreitung der Objektorientierung ist eine Übertragung der Konzepte auf Petrinetze naheliegend und damit die Vorteile der Objektorientierung und die der Petrinetze zu kombinieren.

1.2 Petrinetze

Die von C.A. Petri in [Petri 62] vorgestellten Petrinetze gewinnen immer mehr an Bedeutung. Viele Beispiele für den Einsatz von Petrinetzen zur Beschreibung von Systemen und Algorithmen findet man in der Literatur von Kurt Jensen. In [Jensen 92] wird u.a. ein Kommunikations-Protokoll eines Telefon-Netzwerkes und eine Radarüberwachung vorgestellt. In [Jensen 97] und [Jensen 98] findet man weitere praktische Anwendungen von Petrinetzen, z.B. die Modellierung eines Zugverkehrssystems (Universität Kiel) und die Modellierung der Ausführung von Software für Mobiltelefone (Nokia). Petrinetze werden insbesondere als grafisches Hilfsmittel zur Beschreibung von dynamischen Systemaspekten eingesetzt. Besonders die Möglichkeit der Modellierung paralleler Abläufe machen die Petrinetze zu einer interessanten Technik.

Petrinetze bestehen aus Rechtecken, den Transitionen, die die Aktivitäten darstellen, Ellipsen, den Stellen, die Datenspeicher bzw. Kanäle darstellen und aus Pfeilen, den Kanten, die die Flußrichtung der Marken darstellen. In einem erstellten Netz lassen sich nun Marken, die Daten oder Betriebsmittel darstellen, in die Stellen 'legen'. Diese Marken werden dann bei einer Simulation des Netzes von den Transitionen aus den Eingangsstellen abgezogen und es werden von der Transition neue Marken

generiert, die dann in die Ausgangsstellen ‘gelegt’ werden. In [Jensen 92] werden verschiedene Arten von Netzen vorgestellt. Bei den in dieser Arbeit verwendeten Petrinetzen handelt es sich um gefärbte Petrinetze, wobei durch die Einschränkung der Struktur die objektorientierten Konzepte realisiert werden.

1.3 Minotours: Historie zum Fallbeispiel

In dieser Arbeit wird die objektorientierte Analyse eines Reiseunternehmens als Fallbeispiel mit verschiedenen Ansätzen durchgeführt. Gewählt wurde dieses Beispiel, weil es für ein Reiseunternehmen eine ausführbare Spezifikation mit gefärbten Petrinetzen in [Netzebandt 97] und [Meyer 98] gibt. Die Einarbeitung in dieses Fachgebiet hat also bereits stattgefunden und die Modellierungsansätze können verglichen werden. Für die in [Netzebandt 97] aufgetretenen Probleme werden in dieser Arbeit Lösungsansätze erarbeitet. Bei der Modellierung des Fallbeispiels mit objektorientierten Petrinetzen in Kapitel 5 wird auf die Problemlösungen eingegangen. Aufgrund der fehlenden Werkzeugunterstützung zur Erzeugung von objektorientierten Petrinetzen wird das Reiseunternehmen Minotours etwas eingeschränkt.

Die in [Netzebandt 97] entworfenen Petrinetze wurden mit der Software ‘Design/CPN Version 2.0’ erstellt und simuliert. Alle Aussagen und Anmerkungen in diesem Abschnitt beziehen sich auf diese Version. Die ursprüngliche Beschreibung des Reiseunternehmens ‘Minotours’ wurde von Dave King, University of Brighton, UK, verfaßt und während meines Aufenthaltes in England von mir übernommen und etwas modifiziert.

Minotours ist ein Reiseunternehmen, das Ferienhäuser in Griechenland für einige Wochen im Sommer vermietet. Die Häuser können wochenweise, inklusive Hin- und Rückflug, gebucht werden. Im Büro des Reiseunternehmens werden die Häuser und Flüge in Wandtafeln von den Mitarbeitern verwaltet. Das Ziel der Modellierung des Buchungsvorgangs ist eine ausführbare Spezifikation der Buchungssoftware. Bei dem gewählten Modellierungsansatz wird durch funktionale Dekomposition und evolutionäres Prototyping das Reiseunternehmen ‘Minotours’ mit gefärbten Petrinetzen spezifiziert.

Das gewählte Vorgehen ist ähnlich dem von D. Moldt in [Moldt 96, Kap.8] vorgeschlagenen. Nach einer Zerlegung des Systems in Teilfunktionen, werden diese, jeweils unter einer ablauforientierten Betrachtung mit Hilfe von Szenarien, als gefärbte Petrinetze modelliert. Als Modellierungsgrundlage benötigt der Systementwickler Informationen über das zu entwickelnde System. Um die Schnittstellen des Systems zur Außenwelt zu definieren, wird eine Systemabgrenzung, durch ein Kontextdiagramm wie es aus der strukturierten Systemanalyse bekannt ist ([Yourdon 89]), vorgenommen. Um die Funktionen innerhalb des Systems darzustellen, eignet sich eine funktionale Dekomposition. Hierbei wird das System in Teilfunktionen zerlegt und diese durch Speicher in Beziehung zueinander gesetzt. Für jede Funktion wird eine eindeutige Funktionsbeschreibung angefertigt. Die gewählte Notation kann, wie in [Yourdon 89], eine Mischung aus natürlicher Sprache und Programmiersprache und sowohl textuell, als auch grafisch sein. In Anlehnung an die in [Moldt 96, Kap.8] vorgeschlagene ablauforientierte Sichtweise werden Szenarien als Grundlage für die Modellierung der Petrinetze betrachtet. Ein Szenario ist einer von mehreren erlaubten Handlungsabläufen bei einer gegebenen Anzahl von ausführbaren Funktionen. Es kann sehr viele und sehr komplexe Szenarien geben, insbesondere wenn es viele Ablaufalternativen gibt. Zur Darstellung hat sich eine tabellarische Form bewährt. Das Ziel der Modellierung ist ein Petrinetz, das alle entwickelten Szenarien abdeckt, d.h. die Szenarien müssen mit dem Netz simuliert werden können. Durch schrittweises Erweitern eines zunächst kleinen Petrinetzes, das noch nicht alle gewünschten Szenarien realisiert, wird ein immer komplexeres Netz entwickelt. Die jeweiligen Netze werden als Prototypen bezeichnet, weil diese jeweils simulationsfähig, d.h. ausführbar sind, aber noch nicht die endgültige Systemfunktionalität

umfassen. Jede Erweiterung wird durch die Simulation des Petrinetzes validiert. Das Verfahren ist beendet, wenn ein Prototyp entwickelt wurde, mit dem alle Szenarien simuliert werden können.

Durch die grafische Repräsentation der Petrinetze können Systeme sehr gut beschrieben und bei der Simulation des Systemverhaltens Fehler in der Beschreibung gefunden werden. Anhand der Netze erkennt man die verschiedenen, vom System bereitgestellten Aktivitäten, ihre Funktionsweise und wie sie zueinander in Beziehung stehen. Die Simulation ermöglicht eine interaktive Buchung einer Reise vorzunehmen und illustriert somit den Buchungsvorgang im Gesamtsystem. Prototyping als Vorgehen eignet sich, um während des gesamten Entwicklungsprozesses Änderungsvorschläge des Benutzers zu berücksichtigen. Der Anwender kann verschiedene Szenarien interaktiv durchspielen und Änderungswünsche äußern. Kleine Änderungen können schnell angepaßt werden. Wesentliche Veränderungen des Netzes erfordern eine sehr aufwendige Umprogrammierung, weil die Reihenfolge der Abläufe durch die Form der Netze festgelegt ist. Für Systeme mit sehr vielen Ablaufalternativen erscheint die erprobte Verwendung von gefärbten Petrinetzen zu kompliziert.

In [Meyer 98a] wird eine Architektur zur Entkopplung der Systemfunktionalität und der Benutzeroberfläche vorgestellt. Diese gewährleistet zwar das flexible Austauschen von Elementen der Visualisierung, ändert aber nichts an der starren Struktur der Petrinetze.

Ein vielversprechender Ansatz zur Entwicklung einer ausführbaren Systemspezifikation ist die Übertragung der Konzepte aus der objektorientierten Programmierung auf Petrinetze. Durch die Datenkapselung könnten die, bei der prozeßorientierten Modellierung entwickelten relativ starren, Netze entkoppelt werden. Prototyping als Vorgehensweise ist wegen der genannten Vorteile empfehlenswert.

1.4 Rahmenbedingungen

Die in dieser Arbeit vorgestellten objektorientierten Petrinetz-Ansätze zur Spezifikation von Systemen wurden am Arbeitsbereich ‘Theoretische Grundlagen der Informatik (TGI)’ von Daniel Moldt und Frank Wienberg erarbeitet und stehen im Fokus dieser Untersuchung. Das Gebiet der Petrinetze liegt im speziellen Forschungsinteresse des Arbeitsbereiches. Studien- und Diplomarbeiten zu diesem Thema sind u.a.: [Krauß 96], [Maier 96], [Maier 97] und [Siegel 95].

Ein wichtiger Unterschied zu der Systemanalyse in realen Projekten ist der Punkt der Zusammenarbeit mit dem späteren Benutzer des Systems. Während dort zahlreiche Interviews und Besprechungen stattfinden (sollten), ist dies im Rahmen dieser Arbeit so nicht möglich, da ich selbst das System beschrieben habe. Diskussionen fanden nur universitätsintern mit Kommilitonen und Mitarbeitern statt.

Zur Implementation der Petrinetze wird das Werkzeug Design/CPN 3.02 eingesetzt. Mit diesem Werkzeug können Petrinetze editiert und simuliert werden. Die Verwendung dieser Software hat den Entwurf der Netze stark beeinflußt. Dieser Einfluß wird bei der Vorstellung der objektorientierten Petrinetze erläutert. Da die Modellierung von objektorientierten Petrinetzen nicht unterstützt wird, ist die Entwicklung der Netze sehr aufwendig. Werkzeugaspekte werden ausführlich in einem eigenen Abschnitt erläutert.

Die Verwendung von Word als Texteditor für die Arbeit hat sich wegen der fehlenden Unterstützung zur Einbindung der Petrinetze im Postscript-Format als unbefriedigend erwiesen. Obwohl die Diagramme als eps-Dateien eingebunden werden können, besteht nicht die Möglichkeit diese während des Bearbeiten des Dokumentes zu sehen. Auch die Erzeugung der Netze im eps-Format ist aus der verwendeten Design/CPN Version 3.02 heraus nicht sehr komfortabel. Vor dem Start von De-

sign/CPN muß die Drucker-Umgebungsvariable durch die Befehlszeile ‘setenv DesignPrintCommand "cat > /users/foo/CPNPrintOutput.ps"’ gesetzt werden. Einzelne Seiten werden aus Design/CPN dann als Postscript-Datei gespeichert. Diese müssen umbenannt und mit ‘ps2epsi name.ps‘ in eine eps-Datei konvertiert werden.

Die seit kurzem verfügbare Design/CPN Version 3.1.2 ermöglicht zwar die Erzeugung von Postscript-Dateien der einzelnen Seiten eines Petrinetzes, erfordert aber die exakte Anpassung der Seitenränder an die Diagramme, da diese sonst abgeschnitten werden. Da die Seitenränder nicht durch die Verwendung der Maus angepaßt werden können, ist das Erraten der Seitenlängen in Pixeln zeitaufwendig und muß bei jeder Veränderung der Diagrammgröße berücksichtigt werden.

Bei einigen Diagrammen im Kapitel über die Techniken der Object Modeling Technique und der Unified Modeling Language sind einige der als gestrichelt definierten Linien in den Diagrammen leider im Ausdruck aufgrund des verwendeten Werkzeugs nicht so dargestellt. Aus dem Kontext der textuellen Beschreibung der Abbildungen läßt sich aber auf die referenzierten Kanten schließen.

1.5 Ziele und Aufbau

Das Ziel dieser Arbeit ist die Untersuchung des Einsatzes von objektorientierten gefärbten Petrinetzen im Bereich der Systemspezifikation. Es erfolgt die Betrachtung der Netze als Modellierungstechnik und die Erprobung einer Vorgehensweise bei der Entwicklung der Systemspezifikation.

In Kapitel 2 erfolgt eine Einführung in die Konzepte der objektorientierten Analyse. Dazu werden aus historischen Gründen zuerst die Techniken der Object Modeling Technique (OMT) betrachtet und danach die standardisierte Notation der Unified Modeling Language (UML).

In Kapitel 3 erfolgt die objektorientierte Analyse des als Fallbeispiel gewählten Reiseunternehmens mit OMT und UML. Beim Einsatz der Object Modeling Technique werden die unterschiedlichen Diagramme des statischen, dynamischen und funktionalen Modells entwickelt. Besonders ausführlich erfolgt die Betrachtung der Entwicklung des Objektmodells, da sich dieses mit zunehmender Erfahrung bei der Modellierung mehrfach verändert hat. Bei der Verwendung der Unified Modeling Language werden die Erfahrungen aus der Analyse mit OMT berücksichtigt.

In Kapitel 4 werden verschiedene Ansätze für die Realisierung von objektorientierten Konzepten für Petrinetze erläutert. Es wird gezeigt, wie Klassen, Objekte, Attribute, Methoden, Nachrichten, Assoziationen, Vererbung und Polymorphismus mit Petrinetzen dargestellt werden. Vorgehensweisen zum Einsatz der vorgestellten Petrinetze werden erläutert und auf das zur Modellierung eingesetzte Werkzeug wird eingegangen.

In Kapitel 5 wird das als Fallbeispiel dienende Reiseunternehmen mit objektorientierten Petrinetzen unter Verwendung des in Kapitel 4 vorgestellten Interface-Ansatzes modelliert. Die im Rahmen dieser Arbeit vorgeschlagene Vorgehensweise wird ausführlich dokumentiert. Das Ergebnis der Modellierung ist eine ausführbare Spezifikation des Reiseunternehmens, soweit dieses in diesem Rahmen betrachtet wird. Durch die Simulation können verschiedene Szenarien interaktiv validiert werden. Außerdem werden in diesem Kapitel Modellierungsentscheidungen diskutiert.

Abschließend werden nach der Zusammenfassung der Ergebnisse dieser Arbeit im Ausblick mögliche Themen für weitere Projekte aufgezeigt. Hier werden Themen betrachtet, die im unmittelbaren Zusammenhang mit den Erfahrungen bei dem Einsatz der Petrinetze zur objektorientierten Analyse gemacht wurden, aber aufgrund des Umfangs nicht mehr ausführlich oder gar nicht bearbeitet werden können.

2 OBJEKTORIENTIERTE ANALYSETECHNIKEN

In diesem Kapitel erfolgt die Betrachtung der für diese Arbeit relevanten Aspekte der Object Modeling Technique (OMT) und der Unified Modeling Language (UML). Das wichtigste Konzepte der Objektorientierung ist die Zusammenfassung von Daten und den Operationen, die auf diesen Daten arbeiten. Ein weiteres Konzept ist das der Vererbung. Dieses Konzept bietet die Möglichkeit Hierarchien von Klassen so zu bilden, daß bei der Definition einer Subklasse alle Merkmale der Oberklasse(n), also Datenstrukturen und die in sogenannten Methoden definierte Verarbeitungslogik, übernommen werden. Diese können ggf. erweitert werden. Polymorphie tritt bei der Implementation von geerbten Methoden auf, wenn gleichbenannte Methoden für verschiedene Datentypen definiert sind. Das Hauptziel, das man mit Objektorientierung erreichen möchte, ist eine bessere Wiederverwendbarkeit des einmal geschriebenen Codes, daraus folgend kürzere Entwicklungszeiten und leichtere Wartung. Die Entwicklung von Software besteht u.a. aus Analyse, Design und Implementation. Betrachtet wird in diesem Kapitel der Bereich der Analyse, bei der es das Ziel ist, ein Modell des geplanten Systems zu erstellen, daß das geplante Systemverhalten spezifiziert.

2.1 Object Modeling Technique

In [Rumbaugh et al. 91] mit dem Titel 'Object-Oriented Modeling and Design' wird die 'Object Modeling Technique (OMT)' zur Analyse, zum Design und zur Implementation von Systemen beschrieben. Ausgehend von einer Problembeschreibung des zu entwerfenden Systems werden Objekte identifiziert. Diese Objekte definieren die Datenstrukturen und das Verhalten des Systems. Ein System wird mit der Object Modeling Technique durch drei Modelle beschrieben, das Objektmodell, das dynamische Modell und das funktionale Modell.

2.1.1 Historie

Aus der Zusammenarbeit von James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy und William Lorensen entstand die Object Modeling Technique zur Analyse und zum Design von Softwaresystemen. Objekte der realen Welt werden modelliert und das aus diesen Objekten bestehende Modell als Basis für einen sprachunabhängigen Systementwurf verwendet. In [Rumbaugh et al. 91] wird dieser Ansatz vorgestellt.

2.1.2 Objektmodell

Das Objektmodell beschreibt anhand eines Objektdiagramms, das Objekte und deren Beziehungen zueinander darstellt, die statische Struktur eines Systems. Die Objekte bzw. Objektklassen und deren Beziehungen zueinander werden aus der Problembeschreibung gewonnen. Besonders die später im Fallbeispiel verwendeten Elemente des Objektmodells werden nun eingehender behandelt.

2.1.2.1 Klassen und Objekte

Eine Klasse bzw. Objektklasse beschreibt eine Gruppe von Objekten, mit gleichen Eigenschaften und gleichem Verhalten. Klassen sind Erzeugungsmuster für Objekte. Eine Klasse wird durch einen Namen, Attribute (Eigenschaften) und Operationen (Verhalten) beschrieben. Objekte einer Klasse können zu anderen Objekten anderer Klassen oder zu Objekten der eigenen Klasse in Beziehung stehen. Diese Beziehung ist für alle Objekte einer Klasse gleich. Alle Objekte einer Klasse haben die gleiche

Struktur und das gleiche Verhalten. Die Abbildung 1 zeigt die allgemeine Notation für Klassen.

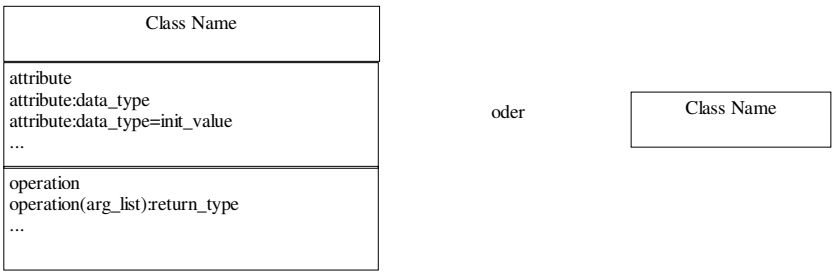


Abbildung 1: Klassendarstellung

Die Objekte des Objektmodells sind Instanziierungen von Objektklassen. Ein Objekt hat eine individuelle Identität und benötigt daher keinen künstlichen (eindeutigen) Bezeichner. Beschrieben wird ein Objekt durch einen Namen, Attribute und Operationen. Die Abbildung 2 zeigt die allgemeine Notation für Objekte.

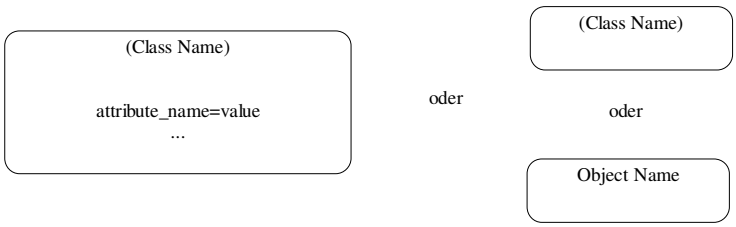


Abbildung 2: Objektdarstellung

Attribute beschreiben die Eigenschaften von Klassen bzw. Objekten. Sie haben einen bestimmten Namen, Datentyp und einen Wert. Bei der Erzeugung eines Objekts kann ein voreingestellter Wert zugewiesen werden. Die Notation wurde schon in den Klassen- und Objektnotationen gezeigt. Die folgenden Kriterien identifizieren unnötige oder falsche Attribute.

- Objekte:
  Die Interpretation eines Attributs als Objekt wird durch das jeweilige System bestimmt. Z.B. ist eine Straße in einer Adreßliste ein Attribut einer Adresse. Bei der Verkehrsplanung könnte eine Straße als eigenes Objekt betrachtet werden.
- Bezeichner:
  Spezielle Objektbezeichner brauchen bei der objektorientierten Sichtweise nicht explizit den Objekten zugewiesen werden. Jedes Objekt wird als einzigartig betrachtet. Auch Objekte derselben Klasse mit den gleichen Attributwerten sind unterschiedlich.
- Verbindungsattribute:
  Gehört ein Attribut eher zu einer Beziehung, als zu einem Objekt, so kann es als Attribut der Verbindung modelliert werden. Ein Beispiel dafür ist das Gehalt eines Mitarbeiters, das Attribut der Beziehung ‘arbeitet für’ zwischen Firma und Mitarbeiter ist.
- Detaillierungsgrad:
  Information, die bei keiner Operation genutzt wird, ist eventuell zu detailliert.

2.1.2.2 Operationen und Methoden

Operationen bestimmen das Verhalten der durch eine Klasse erzeugten Objekte. Operationen können Funktionen oder Transformationen sein. Operationen können polymorph sein, d.h. sie können in unterschiedlichen Klassen unter gleichem Namen auftreten und unterschiedliche Funktionen bzw. Transformationen bewirken. Die Semantik sollte allerdings aus Verständnisgründen erhalten bleiben.

Eine Methode ist die Implementation einer Operation. Operationen bzw. Methoden können parametrisiert sein. Bei jedem Aufruf einer Methode muß die Anzahl und der Typ der Parameter und der Typ des Resultats gleich sein. Die Notation einer Methode ist ein in einer gewählten Programmiersprache geschriebener Code.

2.1.2.3 Assoziationen und Verbindungen

Assoziationen und Verbindungen werden verwendet, um Klassen bzw. Objekte in Beziehung zueinander zu setzen. Eine Assoziation beschreibt eine Gruppe von Verbindungen und besteht zwischen Klassen. Sie kann unidirektional sein, d.h. eine Vorwärts- oder Rückwärtsassoziation, oder bidirektional. Der Assoziationsname definiert die Richtung. Assoziationen entsprechen den Verben oder Verbphrasen in der Systembeschreibung. Allgemein gelten folgende Kriterien zum Auffinden von Assoziationen:

- Physikalischer Ort: neben, Teil von, enthalten in
- Richtungsweisende Aktion: fährt
- Kommunikation: spricht mit
- Besitz: hat, Teil von
- Erfüllung von Bedingungen: arbeitet für, verheiratet mit, verwaltet

Abbildung 3 zeigt die Notation einer Assoziation als Linie zwischen zwei Klassen mit optionaler Beschriftung. Assoziationen können auch zwischen mehr als zwei Klassen bestehen. Die Modellierung von Assoziationen als Klassen, Verwendung von Rollennamen, Sortierung und Qualifizierung ist in [Rumbaugh et al. 91] nachzulesen.

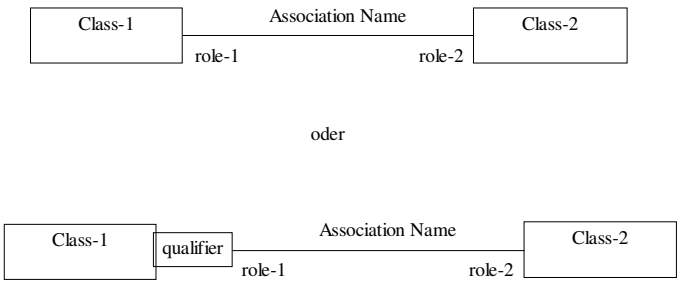


Abbildung 3: Assoziationen zwischen Klassen

Es kann vorkommen, daß unnötige oder falsche Assoziationen gefunden werden. Die folgende Liste von Kriterien hilft beim Eliminieren dieser Assoziationen.

- Assoziationen zwischen eliminierten Klassen:
  Wird eine Klasse, die in einer Assoziation zu einer anderen Klasse steht, eliminiert, so existiert auch die Assoziation nicht mehr.

- Irrelevante Assoziationen:
  Eliminiert werden alle Assoziationen, die sich für das System als irrelevant herausstellen.
- Aktionen:
  Eine Assoziation soll eine strukturelle Eigenschaft des Anwendungsgebietes beschreiben, keine vergänglichen Ereignisse.
- Dreifach-Assoziationen:
  Die meisten Mehrfach-Assoziationen können in Zweifach-Assoziationen geändert oder qualifiziert werden.
- Abgeleitete Assoziationen:
  Abgeleitete Assoziationen brauchen nicht explizit aufgeführt zu werden.
- Schlecht benannte Assoziationen:
  Die Assoziationsnamen müssen sehr sorgfältig gewählt werden. Schlecht benannte Assoziationen sollten umbenannt werden.
- Rollennamen:
  Rollennamen können verwendet werden, um die Assoziation besser zu verstehen. Rollennamen werden den Enden der Assoziation zugewiesen.
- Qualifizierte Assoziationen:
  Ein Bezeichner (qualifier) wird verwendet, um explizit Objekte auf der 'Viele-Seite' einer Assoziation zu unterscheiden.
- Multiplizität:
  Multiplizität, d.h. insbesondere der Unterschied zwischen Kann- und Muß-Beziehungen, kann definiert werden. Da sich die Multiplizität im Laufe der Analyse jedoch noch verändern kann, sollte nicht zu viel Zeit darauf verwendet werden.

Eine Verbindung ist eine Instanziierung einer Assoziation und verbindet Objekte physikalisch oder konzeptuell. Die Darstellung entspricht der von Assoziationen, nur bestehen Verbindungen zwischen Objekten. Verbindungen können mit Attributen versehen werden, wenn ein Attribut eher zu einer Verbindung gehört, als zu einem Objekt.

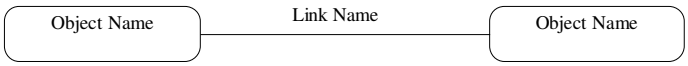


Abbildung 4: Verbindung zwischen Objekten

Multiplizitäten sind eine Notation, mit der Assoziationen versehen werden können, die bestimmen, wieviele Instanziierungen der Klassen, die in einer Assoziation stehen, während einer Instanziierung der Assoziation existieren dürfen bzw. müssen. Hierzu können an jedem Ende der Assoziation Angaben gemacht werden, die die dort befindliche Klasse betreffen. Man unterscheidet die in Abbildung 5 gezeigten verschiedenen Notationsmöglichkeiten. Eine '1' zu '0 oder 1' Assoziation zwischen den Klassen A und B bedeutet, daß es zu einem Objekt der Klasse A ein Objekt der Klasse B geben kann und zu einem Objekt der Klasse B genau ein Objekt der Klasse A existieren muß.

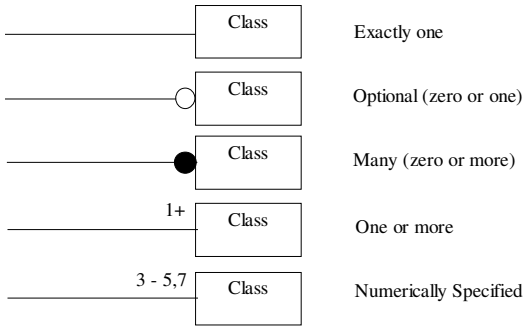


Abbildung 5: Multiplizitäten

Die Aggregation ist eine spezielle Form der Assoziation und bezeichnet eine ‘besteht aus’ oder ‘ist Teil von’ Beziehung. Diese Relation ist transitiv, d.h., wenn A ein Teil von B ist und B ein Teil von C, so ist auch A ein Teil von C. Außerdem ist sie antisymmetrisch, d.h., wenn A Teil von B ist, kann B nicht Teil von A sein. C wird als aggregierendes Objekt bezeichnet, B als aggregierendes und aggregiertes und A als aggregiertes Objekt. Dargestellt ist die Aggregation in Abbildung 6. Objekte vom Typ ‘Assembly Class’ sind aggregierend und Objekte vom Typ ‘Part-1-Class’ und ‘Part-2-Class’ werden aggregiert. Ein Aggregat kann zusätzliche Funktionen erbringen, die von den einzelnen Teilen nicht erbracht werden. Ein Objekt kann von mehreren aggregierenden Objekten aggregiert werden.

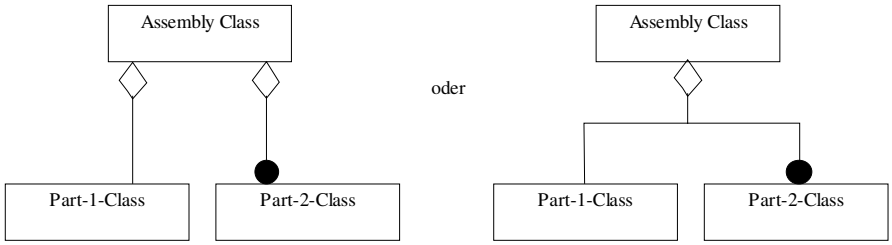


Abbildung 6: Aggregation

2.1.2.4 Vererbung

Das Konzept der Vererbung bietet eine Möglichkeit, Gemeinsamkeiten von Klassen übersichtlich darzustellen. Gleichzeitig werden die Unterschiede von Klassen deutlich. Wird bei der Implementierung eine objektorientierte Programmiersprache verwendet, so können diese Vererbungsbeziehungen direkt umgesetzt werden. Das Konzept der Vererbung erhöht die Wiederverwendbarkeit von Klassen.

Klassen, die in einer Vererbungsbeziehung stehen, unterteilt man in Oberklassen und Unterklassen. Die Oberklasse enthält die Attribute und Operationen, die alle Klassen gemeinsam haben, die in dieser Vererbungshierarchie stehen. Die Unterklassen erben diese Attribute und Operationen von der Oberklasse. Weitere Attribute und Operationen können den Unterklassen hinzugefügt werden. Eine Vererbungshierarchie sollte aus Übersichtlichkeitsgründen nicht mehr als zwei bis fünf Ebenen haben. Die allgemeine Darstellungsweise zeigt die Abbildung 7.

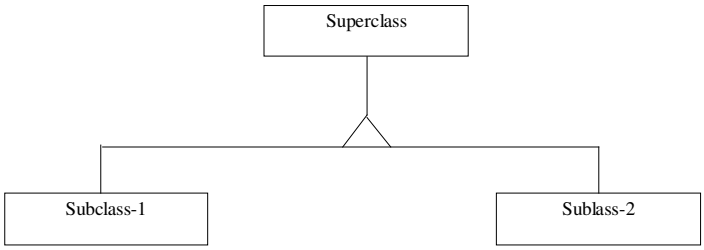


Abbildung 7: Vererbung

Die in einer Oberklasse festgelegten Attribute und Operationen können in den Unterklassen überschrieben, d.h. neu definiert, werden. Nach [Rumbaugh et al. 91] darf bezüglich der Attribute der vordefinierte Wert überschrieben werden, nicht jedoch Name und Typ. Bei den Operationen darf die Methode, d.h. die Implementation, geändert werden, nicht jedoch der Name, Anzahl und Typ der Parameter und der Rückgabetyp. In der Oberklasse kann die Operation durch '{abstract}' gekennzeichnet sein. Das bedeutet, daß die Operationen nur in den Unterklassen implementiert werden. Von der Oberklasse kann es keine Instanziierung geben, sobald eine Operation als '{abstract}' definiert ist.

2.1.2.5 Iteration der Objektmodellierung

Da ein Objektmodell selten nach dem ersten Entwurf korrekt ist, ist es notwendig, mehrere Durchgänge bei der Entwicklung des Modells iterativ zu durchlaufen. Die folgende Aufzählung nennt die Kriterien zur Überprüfung des bisher erstellten Modells.

- Zeichen fehlender Objekte:
  - Asymmetrien in Assoziationen und Generalisierungen
- Zeichen unnötiger Klassen:
  - keine Attribute, Operationen oder Assoziationen
- Zeichen fehlender Assoziationen:
  - fehlende Ausführungspfade für Operationen
- Zeichen überflüssiger Assoziationen:
  - redundante Information in den Assoziationen
- Zeichen von inkorrekten Assoziationen:
  - Rollennamen die zu allgemein oder zu detailliert sind für ihre Klasse

2.1.3 Dynamisches Modell

Das dynamische Modell zeigt anhand von Zuständen und Ereignissen die zeitliche Reihenfolge der Operationsaufrufe. Es besteht aus Szenarien, Event Traces und Zustandsdiagrammen. Im folgenden werden diese und andere benötigte Begriffe erläutert.

2.1.3.1 Ereignisse

Als Ereignisse bezeichnet man die Stimuli, die ein Objekt auf ein anderes bewirkt. Die folgenden Vorkommnisse werden als Ereignisse bezeichnet: Signale, Eingaben, Entscheidungen, Interrupts, Transitionen und Aktionen. Interne Berechnungen sind keine Ereignisse. Ein Ereignis findet zu einem Zeitpunkt statt und hat keine Dauer. Mehrere Ereignisse können gleichzeitig stattfinden. Es können Ereignisklassen definiert werden, die Erzeugungsmuster für Ereignisse sind. Die erzeugten Ereignisse

haben gemeinsame Struktur, Verhalten und Attribute. Die Ereignisse werden in Event Traces dargestellt (s.u.).

2.1.3.2 Benutzerschnittstelle

Bei der dynamischen Modellierung steht die Kontrollogik der Anwendung im Vordergrund, nicht das Design der Benutzerschnittstelle. Dennoch kann das Skizzieren eines GUI (Graphical User Interface) hilfreich sein, um nicht wichtige Dinge zu vergessen. Werkzeuge zur schnellen Erstellung von Oberflächen sollten eingesetzt werden, um diese als Diskussionsgrundlage verwenden zu können.

2.1.3.3 Szenarien

Szenarien werden verwendet, um die Verwendung des Systems durch einen Systembenutzer zu beschreiben. Ein Szenario besteht aus einer Ereignissequenz einer Systemnutzung, d.h. es werden Ereignisse betrachtet, die jeweils das System oder den Benutzer aktiv werden lassen. Ein Ereignis kann Parameter haben. Hat es keine, so wird es als Signal bezeichnet.

Für ein System kann es sehr viele Szenarien geben. Szenarien können als Protokolle zur gedanklichen Systemanwendung betrachtet werden. Sie dienen zum Finden von Ereignissen und somit zur Definition der nötigen Operationen. Wichtig bei der Analyse ist, die für die Analyse sinnvollen Szenarien herauszufiltern. Es können also leicht irrelevante oder zu wenig Szenarien betrachtet werden. Die Diskussion mit dem Auftraggeber führt zu der Betrachtung der relevanten Szenarien. Abbildung 8 zeigt ein Beispiel für ein Szenario für den Vorgang eines Telefonates aus [Rumbaugh et al. 91].

caller lifts receiver
dial tone begins
caller dials digit (4)
dial tone ends
caller dials digit (5)
caller dials digit (3)
caller dials digit (7)
called phone begins ringing
ringing tone appears in calling phone
called party answers
called phone stops ringing
ringing tone disappears in calling phone
phones are connected
called phone hangs up
phones are disconnected
caller hangs up

Abbildung 8: Szenario

2.1.3.4 Event Traces

Ein Event Trace wird in einem Event Trace Diagramm dargestellt, das Objekte und Ereignisse in Beziehung zueinander setzt. Die Objekte werden als vertikale Linien gezeichnet und die Ereignisse als horizontale gerichtete Kanten vom Sender zum Empfänger zwischen ihnen. Die Zeitachse verläuft von oben nach unten. Durch ein Event Trace Diagramm läßt sich somit grafisch die zeitliche Reihenfolge des Auftretens von Ereignissen darstellen und wie sich die Objekte verhalten. Auch konkurrente Ereignisse können dargestellt werden. Abbildung 9 zeigt das Event Trace Diagramm für das

Szenario-Beispiel aus Abbildung 8.

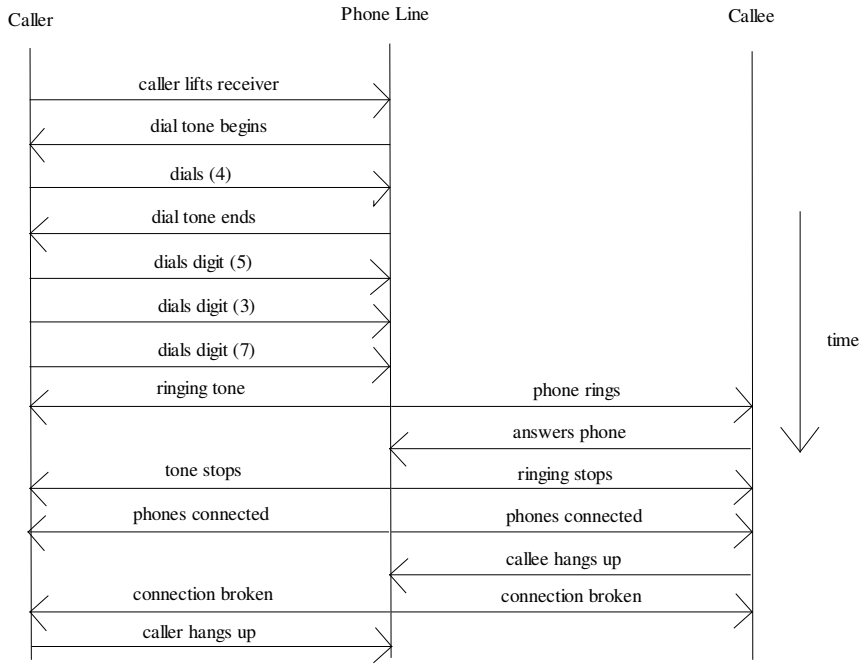


Abbildung 9: Event-Trace

2.1.3.5 Zustände

Ein Zustand eines Objekts ist eine Wertbelegung seiner Attribute. Ereignisse können eine Wertbelegung, d.h. einen Zustand eines Objekts ändern. Je nach dem Zustand kann ein Objekt unterschiedlich auf Ereignisse reagieren. Eine Reaktion kann aus einer Aktion und einer Zustandsänderung bestehen. Einem Zustand wird eine Zeitdauer zugeschrieben. Charakterisiert wird ein Zustand u.a. durch eine Ereignissequenz, die in diesen Zustand führt und durch die Ereignisse, die in diesem Zustand akzeptiert werden.

2.1.3.6 Zustandsdiagramme

Zustandsdiagramme setzen Ereignisse und Zustände in Beziehung zueinander. Zustände werden als abgerundete Rechtecke dargestellt, die Ereignisse als gerichtete Kanten zwischen den Zuständen. Sie werden als Transitionen bezeichnet. Die Kanten sind mit dem Ereignisnamen beschriftet. Unterschiedliche Transitionen, ausgehend von einem Zustand, enden in unterschiedlichen Zuständen. Somit werden alle möglichen Zustandsfolgen eines Objekts durch die Transitionen in einem Diagramm dargestellt. Der Startzustand wird mit einem gefüllten Kreis und eingehendem Pfeil gekennzeichnet. Wird ein Objekt nach einer Zustandsfolge eliminiert, so wird dies durch Erreichen eines eingekreisten ausgefüllten Kreises dargestellt.

Unterschiedliche Diagramme können über gemeinsame Ereignisse interagieren. Ein Zustandsdiagramm kann als Erzeugungsmuster für Szenarien betrachtet werden. Ein Szenario ist somit eine Instanziierung eines Zustandsdiagramms. Transitionen können mit Bedingungen versehen werden, die erfüllt sein müssen, um das Ereignis stattfinden zu lassen.

Als Kontrolloperationen gibt es Aktivitäten und Aktionen. Aktivitäten sind mit einem Zustand asso-

ziiert und haben eine Zeitdauer. Sie starten mit betreten des Zustands und enden nach Abarbeitung oder durch Verlassen des Zustands. Aktionen sind mit Ereignissen assoziiert und werden als zeitlos betrachtet. Sie werden hinter dem Ereignisnamen als Beschriftung der Transition mit aufgeführt. Die Abbildung 10 zeigt eine Zusammenfassung der Elemente eines Zustandsdiagramms.

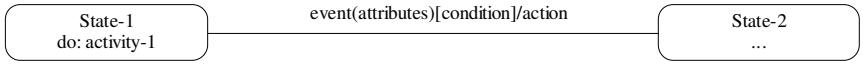


Abbildung 10: Zustandsdiagramm

Zustandsdiagramme können strukturiert werden. Ereignisse und Zustände werden in Hierarchien organisiert, mit Möglichkeiten der Vererbung von Struktur und Verhalten. Beispiele für Strukturierung und Vererbung findet man in [Rumbaugh et al. 91]. Die Abbildung 11 zeigt die Notation für Nebenläufigkeit.

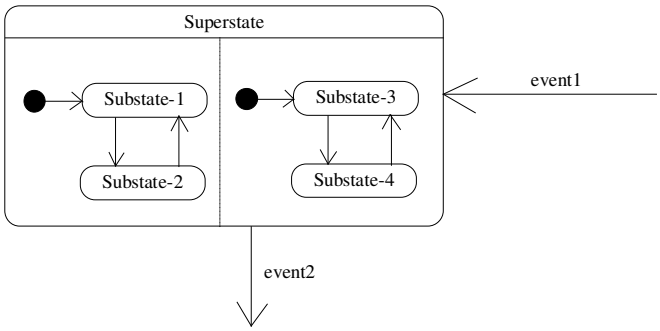


Abbildung 11: Strukturiertes Zustandsdiagramm

2.1.4 Funktionales Modell

Im funktionalen Modell wird durch Datenflußdiagramme, die Prozesse und Datenflüsse darstellen, beschrieben, welche Berechnungen im System ausgeführt werden. Wann die Berechnungen stattfinden ist Teil des dynamischen Modells.

2.1.4.1 Datenflußdiagramme

Datenflußdiagramme zeigen die Beziehungen zwischen Ein-und Ausgabewerten. Prozesse stellen den Datenfluß zwischen Objekten dar. Diese transformieren Daten. Akteure sind Objekte, die Daten produzieren oder konsumieren. Datenspeicher sind passive Elemente, die zur persistenten Datenspeicherung verwendet werden. Die einzelnen Elemente von Datenflußdiagrammen werden nun näher erläutert.

2.1.4.2 Prozesse

Prozesse transformieren Werte. Dies können Berechnungen sein, das Lesen und Schreiben von Dateien oder das Darstellen von Grafiken. Dargestellt werden Prozesse wie in Abbildung 12 als Ellipsen, die mit dem Prozeßnamen beschriftet sind. Eingehende gerichtete Kanten sind mit den Eingabetypen, ausgehende gerichtete Kanten mit den Ausgabetypen beschriftet. Die Prozeßbeschreibung findet man an anderer Stelle als mathematische Notation, natürlichsprachliche Beschreibung, Pseu-

docode o.ä.

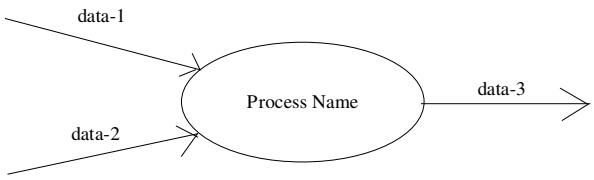


Abbildung 12: Prozeß

2.1.4.3 Datenflüsse

Datenflüsse verbinden Objekte bzw. Prozesse in Datenflußdiagrammen. Sie bezeichnen jeweils die Ein- bzw. Ausgabe. Ein Datenfluß verändert keine Werte. Die Darstellung ist ein mit dem Datentyp beschrifteter Pfeil. Abbildung 13 zeigt die Darstellung von Datenkopien durch sich gabelnde Pfeile, Aufspaltung von aggregierten Daten durch beschriftete Gabelung und Aggregation von Daten durch zusammenführende Pfeile.

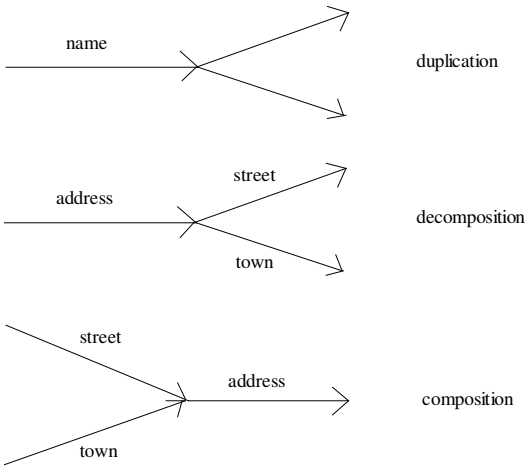


Abbildung 13: Datenfluß

2.1.4.4 Akteure

Akteure sind aktive Objekte, die Daten produzieren oder konsumieren und am Rand von Datenfluß-diagrammen liegen. Sie werden auch als Datenquellen bzw. Datensenken bezeichnet. Dargestellt werden sie durch Rechtecke. Produzenten haben wegführende, Konsumenten hineinführende Kanten.

2.1.4.5 Datenspeicher

Datenspeicher sind passive Objekte und werden zur persistenten Speicherung von Daten verwendet. Die Speicherung erfolgt nach festgelegten Reihenfolgen. Der Zugriff auf gespeicherte Daten kann in beliebiger Reihenfolge stattfinden. Dargestellt werden Datenspeicher in den Datenflußdiagrammen durch zwei horizontale parallele Linien, zwischen denen der Name des Speichers steht. Hineinführende Pfeile verdeutlichen die Möglichkeit des Lesens und des Schreibens von Daten. Aus dem Speicher herausführende Pfeile bedeuten, daß Daten nur gelesen werden. Ein Speicher mit konstanten

Daten hat keine hineinführenden Kanten. Abbildung 14 zeigt einen produzierenden und einen konsumierenden Akteur und einen Prozeß, der Daten in einen Speicher schreibt oder vorhandene Daten ändert.

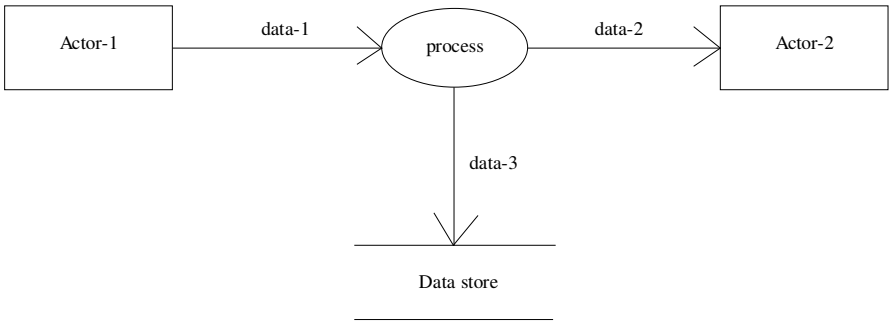


Abbildung 14: Datenspeicher

2.1.4.6 Strukturierung

Komplexe Prozesse können durch verschachtelte Diagramme strukturiert werden. Ein Prozeß wird durch die Einführung einer weiteren Ebene verfeinert dargestellt. Die Verfeinerung und der verfeinerte Prozeß haben die gleichen Ein- und Ausgaben. Die Verfeinerung kann Speicher enthalten, die auf der oberen Ebene nicht sichtbar sind.

2.1.4.7 Kontrollflüsse

Kontrollflüsse sind Teil des dynamischen Modells, können aber manchmal auch im funktionalen Modell sinnvoll sein. Ein Kontrollfluß ist kein Eingabewert für einen Prozeß, sondern ein Wahrheitswert, der innerhalb eines anderen Prozesses erzeugt wurde. Dargestellt wird solch ein Kontrollfluß als gepunktete gerichtete Kante von dem wahrheitswertproduzierenden Prozeß zu dem kontrollierten Prozeß.

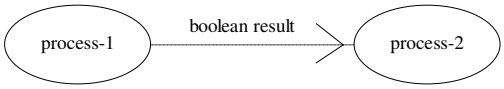


Abbildung 15: Kontrollfluß

2.1.4.8 Operationen

Die Prozesse werden als Operationen in den Objekten definiert. Die Operationen werden den Objekten zugeordnet. Ein Objekt besteht dann, abgesehen von dem Objektnamen und den Attributen, aus den Operationen. Die Operationen können auf verschiedene Arten spezifiziert werden. Zum Beispiel durch mathematische Funktionen, Vor- und Nachbedingungen, Pseudocode, natürliche Sprache o.ä.

Spezifiziert wird die Signatur, also Anzahl und Typ der Argumente und Rückgabewerte, sowie die Wertetransformation, d.h. wie Ausgabewerte aus den Eingabewerten berechnet werden und welche Seiteneffekte auftreten. Die Spezifikation von Operationen bezieht sich auf die von außen sichtbare

Wirkung der Operation und nicht auf interne Details oder eine Implementation. Abbildung 16 aus [Rumbaugh et al. 91] beschreibt beispielhaft die Funktion des Aufbaus einer Telefonverbindung. Die folgenden Punkte unterstützen das definieren von Operationen:

- Operationen vom Objektmodell
  - Lesen/Schreiben von Attributwerten
- Operationen von Ereignissen
  - Jedes an ein Objekt geschickte Ereignis entspricht einer Operation auf diesem Objekt.
  - Die Implementierung durch explizite Methoden.
- Operationen von Zustandsaktionen und Aktivitäten
  - Die Aktionen und Aktivitäten, die als Funktionen beschrieben werden, sollten als Operationen im Objektmodell beschrieben werden.
- Operationen von Funktionen
  - Funktionen aus den Datenflußdiagrammen sollten beschrieben und ins Objektmodell integriert werden.

Function: connect call
Inputs: phone line,number dialed, current settings of switches
Outputs: new settings of switches, connection status
Transformation: Connect the calling phone to the dialed phone by closing connections in the switcher, observing the following constraints.
Constraints: Only two lines at a time may be connected on any one circuit.
Previous connections must not be disturbed.
If the called line is already in use, then no switches are closed, and the status is reported as busy.
If a connection is impossible bacaus too many switches are in use, then no switches are closed, and the status is reported as switcher busy.

Abbildung 16: Funktionsbeschreibung

2.1.5 Vorgehen bei der Analyse

In [Rumbaugh et al. 91] wird ein iteratives Vorgehen bei der Verwendung von OMT zur Analyse und zum Design von Systemen vorgeschlagen. In der Analysephase wird ein Modell des Systems entworfen, das spezifiziert, was das System leisten wird. Das Modell sollte für den Auftraggeber verständlich sein und eine Grundlage zur Kommunikation liefern. Das Ergebnis des Systemdesigns ist ein unter Implementationsgesichtspunkten erstelltes Modell des Systems. Es bietet die Grundlage zur Generierung von effizientem Code. Oft können viele Teile des Analysemodells direkt in das Designmodell ohne Veränderungen übernommen werden. Da die Modelle von Analyse und Design in sehr enger Beziehung zu einander stehen, ist es von großem Vorteil, für die Entwicklung dieser Modelle ein iteratives Vorgehen zu wählen und diese Modelle bei jeder Iteration nebenläufig zu entwikkeln.

Für die Analyse werden die folgenden Aktivitäten vorgeschlagen, um das Objektmodell, das dynamische Modell und das funktionale Modell zu erstellen:

1. Verfassen einer Problembeschreibung, d.h. was entwickelt werden soll
2. Entwicklung des Objektmodells
   - Identifikation von Klassen
   - Erstellen eines Datenwörterbuchs
   - Hinzufügen von Assoziationen zwischen Klassen

- Finden von Attributen für Objekte und Assoziationen
- Einführen von Vererbungsbeziehungen
- Szenarien verwenden, um Ausführungspfade zu überprüfen
- Iteration über die zuvor genannten Aktivitäten, wenn notwendig
- Organisation von Klassen in Modulen

  Objektmodell = Objektdiagramm + Datenwörterbuch

3. Entwicklung des dynamischen Modells
   - Erstellen von Szenarien für typische Abläufe im System
   - Identifikation von Ereignissen zwischen Objekten
   - Erstellen eines Event-Trace-Diagrammes für jedes Szenario
   - Erstellung eines Event-Flow-Diagrammes für das System
   - Entwicklung von Zustandsdiagrammen für Klassen mit wichtigem dynamischem Verhalten
   - Konsistenzüberprüfung der Zustandsdiagramme

     Dynamisches Modell = Zustandsdiagramme + Event-Flow-Diagramm
4. Entwicklung des funktionalen Modells
   - Identifizieren von Ein- und Ausgabewerten
   - Entwicklung von Datenflußdiagrammen zur Darstellung der funktionalen Abhängigkeiten
   - Beschreibung der einzelnen Funktionen
   - Hinzufügen von Bedingungen/Einschränkungen (Constraints)
   - Festlegen von Optimierungskriterien für die Implementation

     Funktionales Modell = Datenflußdiagramme + Constraints
5. Iteration, Überprüfung und Verfeinerung der drei Modelle
   - Integration von wichtigen Operationen in das Objektmodell
   - Prüfen der Konsistenz von Klassen, Assoziationen und Operationen
   - Testen des Modells unter Verwendung der Szenarien
   - Hinzufügen weiterer (untypischer) Szenarien
   - Iteration über alle Modelle zur Vervollständigung des Analysemodells

     Analysedokument =

     Problembeschreibung + Objektmodell + Dynamisches Modell + Funktionales Modell

Zu Beginn des Objektdesigns wird i.a. ein System-Design-Dokument erstellt, das die Systemstruktur auf einem hohen Abstraktionsniveau beschreibt. Beim Objektdesign wird das Analysemodell so erweitert, daß als Ergebnis ein Designdokument vorliegt, das als solide Basis für die Implementation dient. Der Systementwurf ist unabhängig von einer bestimmten Programmiersprache, Datenbank o.ä. Der Prozeß des Objektdesigns umfaßt die folgenden Schritte:

1. Definition der Operationen für das Objektmodell unter Verwendung der anderen Modelle
2. Entwurf von Algorithmen zur Implementation der Operationen
3. Optimierung der Pfade für den Zugriff auf die Daten
4. Überarbeiten der Klassenstruktur zur Erhöhung der Vererbungsbeziehungen
5. Implementation der Assoziationen
6. Bestimmen der Repräsentation der Attribute
7. Aufteilen der Klassen und Assoziationen in Module

   Designdokument = Detailliertes Objektmodell + Detailliertes Dynamisches Modell + Detailliertes Funktionales Modell

2.2 Unified Modeling Language

In diesem Kapitel wird die Unified Modeling Language (UML) Version 1.1 vorgestellt. Nach einem Überblick über die Entwicklung von UML werden die einzelnen Notationen eingeführt. Ein Modell eines Systems besteht aus mehreren Diagrammen. Im Rahmen dieser Arbeit können die Notationen nicht vollständig detailliert erklärt werden. In diesem Kapitel werden die Grundideen der Notationen und Beispiele für die grafische Realisierung aufgeführt. Einige Diagramme sind ähnlich zu denen von OMT. Die Diagramme sind größtenteils aus der englischsprachigen Literatur übernommen.

Eine ausführliche Beschreibung der Unified Modeling Language bieten [UML 1.1 Summary], [UML 1.1 Notation-Guide], [UML 1.1 Semantics], [UML 1.1 Extension for Objectory Process for Software Engineering], [UML 1.1 Extension for Business Modeling], [Object Constraint Language Specification]. Während des Schreibens dieser Arbeit ist eine Menge Literatur zum Thema UML erschienen. Wesentliche Veränderungen hat es gegenüber UML 1.1 nicht gegeben, dennoch sei hier erwähnt, daß sich einige Details der in dieser Arbeit verwendeten Notation geändert haben können. U.a. ist folgende Literatur zum Thema UML erschienen: [Burkhardt 97], [Fowler 97], [Kahlbrandt 98], [Larman 98], [Oesterreich 97], [Quatrani 98], [Texel 97]. Weitere Quellen sind im Internet u.a. unter den Adressen ‘www.rational.com’ und ‘www.objectnews.com’ verfügbar. Die verwendeten deutschsprachigen Begriffe sind [ObjektSpektrum 98] entnommen. Einige werden hier auszugsweise vorgestellt: Aggregation = aggregation, Akteur = actor, Aktivität = activity, Assoziation = association, Einschränkung = constraint, Generalisierung = generalization, Interaktionsdiagramm = interaction diagram, Klasse = class, Klassendiagramm = class diagram, Kollaborationsdiagramm = collaboration diagram, Komponente = component, Komposition = composition, Lebenslinie = lifeline, Methode = method, Multiplizität = multiplicity, Nachricht = message, Notiz = note, Oberklasse = superclass, Objekt = object, Objektbeziehung = link, Objektdiagramm = object diagram, Operation = operation, Rolle = role, Schnittstelle = interface, Sequenzdiagramm = sequence diagram, Unterklasse = subclass, Use-Case = use case, Use-Case-Diagramm = use case diagram, Zustandsdiagramm = state chart diagram

2.2.1 Historie

Die Unified Modeling Language ist die Vereinigung der bewährtesten Techniken verschiedener objektorientierter Methoden zur Systementwicklung. Sie dient zur Spezifikation, Visualisierung, Konstruktion und Dokumentation von Softwaresystemen, Geschäftsprozessen und Systemen im allgemeinen.

Motiviert wurde die Entwicklung von UML u.a. dadurch, daß Beschreibungsmöglichkeiten für Modelle von Systemen benötigt werden, um von der Systemkomplexität zu abstrahieren und die Kommunikation zwischen Projektteams und dem Auftraggeber zu verbessern. Mit zunehmender Systemkomplexität wachsen auch die Anforderungen an die Beschreibungsmethoden und -techniken. Softwareentwickler suchen nach Möglichkeiten, um den Softwareentwicklungsprozeß zu automatisieren. Auf dem Markt der objektorientierten Methoden gab es keine führende Methode. Dies hielt die Anwender davon ab, objektorientierte Methoden einzusetzen und Anbieter waren zurückhaltend mit der Entwicklung von Werkzeugen für die objektorientierte Systementwicklung. Die Auswahl der besten Techniken vieler Methoden war eine Möglichkeit einen Standard zu definieren und die Anwendung objektorientierter Systementwicklung zu fördern.

Die Entwickler von UML haben sich als Ziel gesetzt, Notationen mit der notwendigen Semantik zu definieren, um den Ansprüchen der Systementwicklung moderner, komplexer Systeme gerecht zu werden. Die Unified Modeling Language soll schnell zu erlernen sein, Erweiterungen an neue Anforderungen ermöglichen, unabhängig von bestimmten Programmiersprachen und Entwicklungsmetho-

den sein, einen Formalismus zur UML-Definition bieten, den Einsatz objektorientierter Methoden zur Systementwicklung fördern und innovative Entwicklungskonzepte unterstützen.

Begonnen hat die Entwicklung von UML im Oktober 1994 mit der gemeinsamen Arbeit von Grady Booch und James Rumbaugh für die 'Rational Software Corporation'. Da sich ihre Methoden, die Booch-Method und die Object Modeling Technique, unabhängig aufeinander zubewegten, beschlossen sie gemeinsam an einem neuen Ansatz zur Systementwicklung zu arbeiten. Das Ergebnis war im Oktober 1995 die Unified Method 0.8. Zum Jahreswechsel begann Ivar Jacobson, u.a. Autor von [Jacobson 92], für Rational zu arbeiten und entwickelte mit Booch und Rumbaugh die Unified Modeling Language 0.9 zum Juni 1996. Die Version 0.91 erschien im Oktober 1996 und die Object Management Group (OMG) stellte ein 'Request for Proposal'.

Außer der Rational Software Corporation machten die folgenden Firmen Vorschläge zur Entwicklung von UML: Microsoft Corporation, Hewlett-Packard Company, Oracle Corporation, Sterling Software, MCI Systemhouse Corporation, Unisys Corporation, ICON Computing, IntelliCorp, i-Logix, IBM Corporation, ObjecTime Limited, Platinum Technology Inc, Ptech Inc, Taskon A/S, Reich Technologies und Softeam.

Im Januar 1997 erschien die Version 1.0 und nach weiterer Überarbeitung im Herbst 1997 die Version 1.1, die dieser Arbeit als Grundlage dient. Die Abbildung 17 zeigt eine Übersicht über die Entwicklung der Unified Modeling Language aus [UML 1.1 Summary]. Dargestellt ist neben der zeitlichen Entwicklung auch der Einfluß anderer Methoden. Deutlich erkennbar ist auch der Anspruch der Kommerzialisierung von UML.

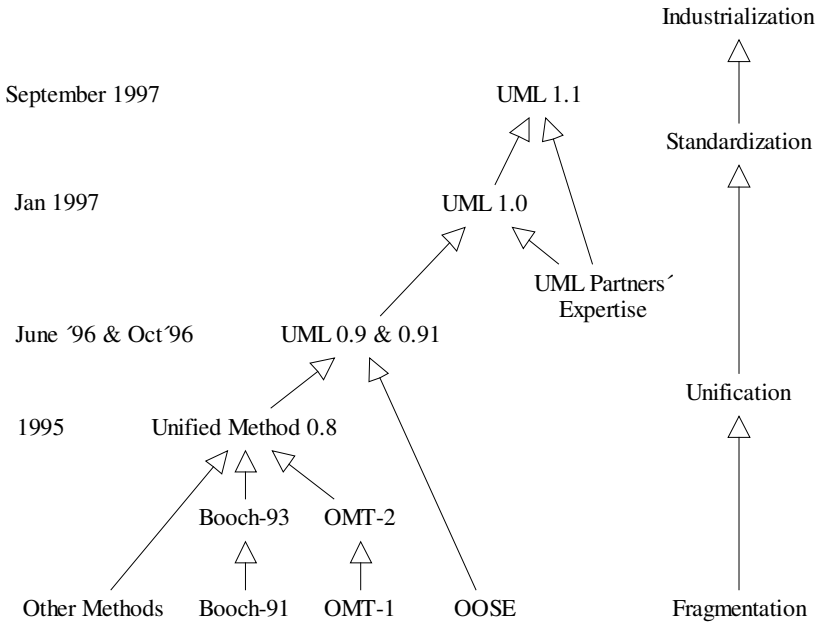


Abbildung 17: Entwicklung von UML

2.2.2 Klassendiagramm

Ein Klassendiagramm stellt die statische Struktur eines Systems dar. Es enthält Objektklassen und deren Beziehungen (Assoziationen) zueinander. Klassendiagramme zeigen keine temporären und dynamischen Aspekte, obwohl solche Informationen implizit in den Klassen und Assoziationen enthalten sein können. Klassendiagramme können in Paketen organisiert sein.

2.2.2.1 Klassen

Eine Klasse beschreibt eine Menge von Objekten mit gleicher Struktur, Verhalten und Beziehungen. Dargestellt werden Klassen durch Rechtecke mit drei Bereichen, für den Namen, die Attribute und die Operationen (Methoden). Abbildung 18 zeigt die allgemeine Notation für Klassen. Die Abschnitte mit den Attributen und Operationen brauchen nicht angezeigt zu werden. Werden sie angezeigt, sind aber leer, dann gibt es bei der entsprechenden Klasse keine Attribute bzw. Operationen. Der Modellierer kann außer den vordefinierten Abschnitten für Attribute und Operationen eigene Abschnitte definieren.

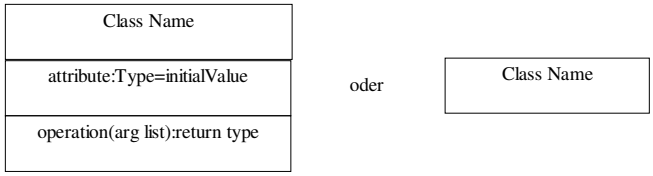


Abbildung 18: Klassennotation

2.2.2.2 Attribute

Die Syntax für die Attribute (attributs) ist '`visibility name:type-expression = initial-value {property-string}`', wobei die visibility '+' für public, '-' für private oder '#' für protected sein kann. Weitere Details entnehme man [UML 1.1 Notation-Guide].

2.2.2.3 Operationen

Eine Operation ist ein Service, den ein Objekt ausführen kann. Eine Operation hat einen Namen und eine Argumentliste. Die Syntax für eine Operation ist '`visibility name(para- meter-list):return-type-expression{property-string}`'. Die Parameterliste hat die Form '`kind name:type-expression = default-value`'. Die Spezifikation von Operationen kann in Notizen im Diagramm als Einschränkung oder Kommentar enthalten sein. Im folgenden werden Operationen auch als Methoden bezeichnet.

2.2.2.4 Schnittstellen

Schnittstellen (interfaces) sind spezielle Elemente, die die nach außen sichtbaren Methoden von Klassen enthalten. Ein Interface hat keine Attribute und von einem Interface gibt es keine Implementation. Wie in Abbildung 19 dargestellt, wird das Keyword «`interface`» zur Kennzeichnung verwendet oder es wird als Kreis dargestellt. Neben dem Kreis wird der Name des Interface notiert. Methoden werden bei dieser Notation nicht angegeben. Wird ein Interface in einem System verwendet, so muß eine Klasse definiert werden, die dieses Interface inklusive aller Methoden implementiert. Von dieser Klasse kann es dann instanziierte Objekte geben. Der Sinn von Interfaces ist, bestimmte Klassen zu definieren, die in allen Systemen die gleiche Funktionalität aufweisen (müssen), auf die sich der Entwickler verlassen kann. Von der eigentlichen Implementation der Methoden wird abstrahiert.

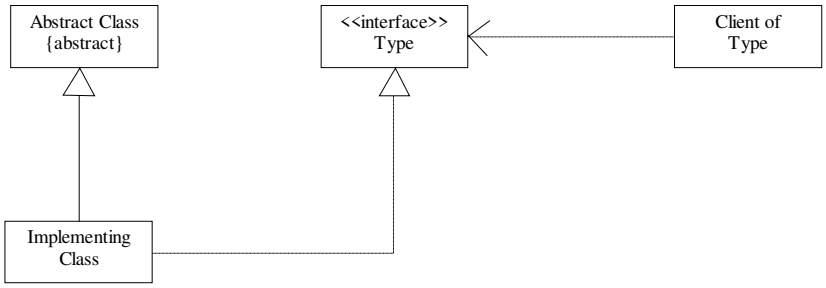


Abbildung 19: Schnittstellendarstellung

2.2.2.5 Utility

Um Attribute und Operationen global zu deklarieren werden spezielle Klassen verwendet. Dieser Klassentyp heißt Utility. Die Abbildung 20 zeigt ein Beispiel für eine Utility-Klasse.

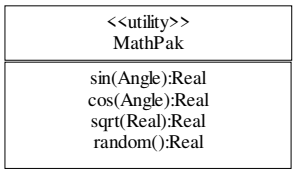


Abbildung 20: Utility

2.2.2.6 Assoziationen

Assoziationen beschreiben Beziehungen zwischen Klassen. Die unterschiedlichen Elemente bei der Notation von Assoziationen werden im folgenden eingeführt. Die Abbildung 21 zeigt die allgemeine Notation für Assoziationen. Eine binäre Assoziation verbindet genau zwei Klassen. Das schließt auch die Selbstreferenzierung einer Klasse ein. Dargestellt wird sie als Linie, die mit dem Assoziationsnamen beschriftet sein kann. Der Assoziationsname kann mit einem kleinen ausgefüllten Dreieck kombiniert werden, um die Leserichtung anzugeben.

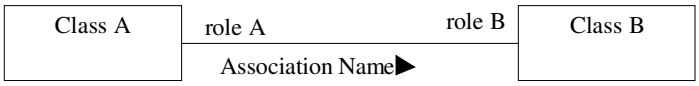


Abbildung 21: Assoziation

Zwischen Klassen kann es mehrere Assoziationen geben, von denen nur eine zu einem Zeitpunkt instanziiert sein darf. Diese Art Assoziation wird durch die Einschränkung '{or}' gekennzeichnet. Assoziationen haben mindestens zwei Enden. Bei den Assoziationsenden können verschiedene Angaben stehen. Den Enden einer Assoziation kann eine Rolle zugeschrieben werden. Multiplizität wird verwendet, um die Kardinalität für eine Klasse bezüglich einer Rolle in einer Assoziation anzugeben. Die Schreibweise für die Multiplizitätsangabe ist in der Abbildung 22 dargestellt. Weitere Angaben können sein: qualifier, navigability, aggregation indicator, interface specifier, changeability und visibility. Die Bedeutung entnehme man der UML-Literatur.

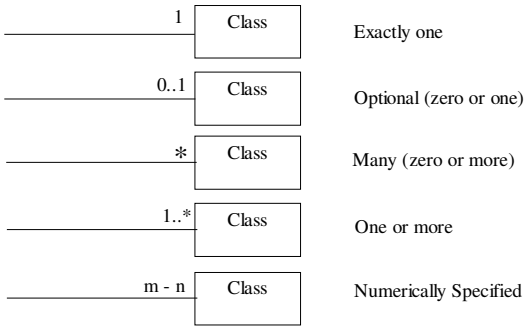


Abbildung 22: Multiplizität

Eine Assoziationsklasse ist eine Assoziation mit Klasseneigenschaften, also mit Attributen und Operationen. Dargestellt wird eine Assoziationsklasse als Klassensymbol, das durch eine gestrichelte Linie mit einer Assoziation verbunden wird. Man betrachte hierzu die Abbildung 23. Ein bekanntes Beispiel ist die Beziehung zwischen einem Angestellten und dem Arbeitgeber. Statt das Gehalt eines Angestellten als Attribut in der Klasse 'Angestellter' zu modellieren, wird eine Assoziationsklasse 'Angestellt bei' hinzugenommen, die als Attribut das Gehalt hat.

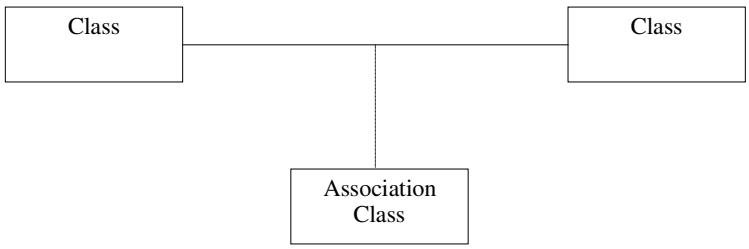


Abbildung 23: Assoziationsklasse

2.2.2.7 Aggregation und Komposition

Durch Aggregation werden zusammengesetzte Klassen definiert, die aus anderen Klassen bestehen. Zur Existenz eines zusammengesetzen Objekts brauchen aber nicht alle Bestandteile gleichzeitig zu existieren. Abbildung 24 zeigt die Notation für die Aggregation. Eine detaiilierte Erklärung der Aggregation findet man im OMT-Kapitel.

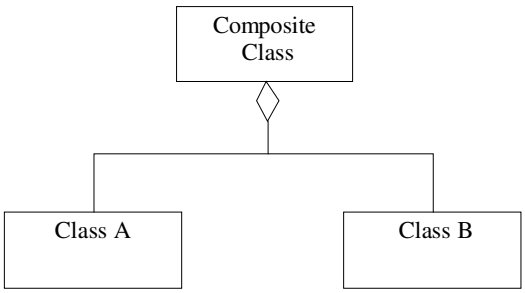


Abbildung 24: Aggregation

Als Komposition wird eine sehr strenge Form der Aggregation bezeichnet. Bei der Komposition

dürfen die Teile nur zu einem Ganzen gehören und sie existieren nur solange, wie das Ganze existiert. Die Komposition ist in Abbildung 25 dargestellt. Teile einer Komposition können, im Gegensatz zur Aggregation, nicht gleichzeitig Bestandteil mehrerer Objekte sein.

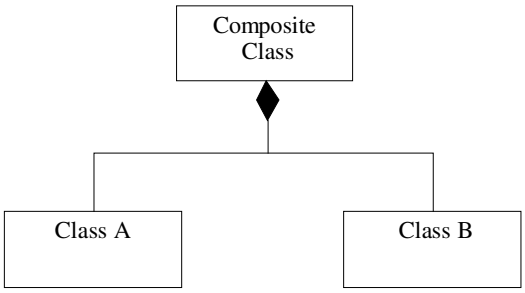


Abbildung 25: Komposition

2.2.2.8 Vererbung

Vererbung (inheritance) ist ein Konzept, das Oberklassen und Unterklassen definiert und in Beziehung zueinander setzt. Unterklassen sind vom Typ einer Oberklasse und enthalten i.a. noch weitere Information. Bei der Vererbung spricht man auch von Generalisierung und Spezialisierung. Die Abbildung 26 zeigt die allgemeine Darstellung einer Vererbungsbeziehung. Die Unterklassen erben gemeinsame Eigenschaften und gemeinsames Verhalten von der Oberklasse.

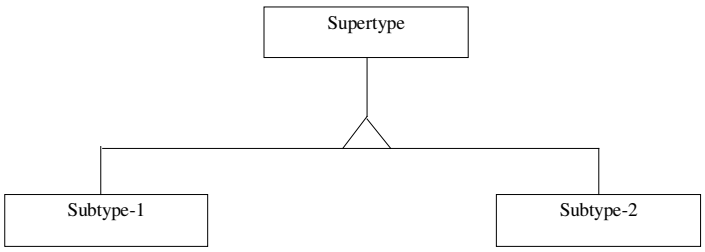


Abbildung 26: Vererbung

2.2.3 Objektdiagramm

Ein Objektdiagramm ist eine Instanz eines Klassendiagramms. D.h. ein Klassendiagramm ist das Erzeugungsmuster für Objektdiagramme. In diesem haben die Instanzen der Klassen (die Objekte) und Instanzen der Assoziationen (die Objektverbindungen) konkrete Werte. Ein Objektdiagramm ist eine Momentaufnahme eines Systemzustands zu einem bestimmten Zeitpunkt. Verwendet werden Objektdiagramme eher selten und werden hier nur der Vollständigkeit halber erwähnt.

2.2.3.1 Objekte

Ein Objekt ist eine Instanz einer Klasse und hat eine Identität und Attributwerte. Die Darstellung ist ähnlich der von Klassen. Im oberen Teil ist der Name und die erzeugende Klasse durch '`objectname:classname`' angegeben. Die zugehörige Klasse und die Attribute brauchen nicht angezeigt zu werden. Abbildung 27 zeigt die Notationsmöglichkeiten für Objekte. Verwendet wird

diese Notation in Objekt-, Interaktions- und Kollaborationsdiagrammen.

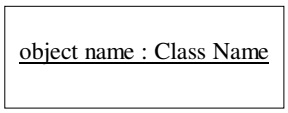


Abbildung 27: Objekt

2.2.3.2 Objektverbindungen

Eine Objektverbindung ist eine Instanz einer Assoziation. Multiplizität ist nicht vorhanden, weil eine Verbindung eine Instanz einer Assoziation ist.

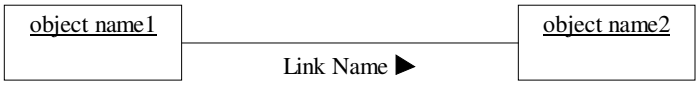


Abbildung 28: Verbindung zwischen Objekten

2.2.4 Use-Cases

Use-Cases beschreiben die für einen externen Systembenutzer sichtbare Funktionalität eines Systems. Der Benutzer wird als Akteur bezeichnet. Ein Use-Case-Diagramm zeigt die Beziehungen zwischen Akteuren und Use-Cases im modellierten System. Auch andere Systeme werden als Akteure modelliert, wenn sie mit dem durch die Use-Cases abgegrenzten System in einer Beziehung stehen. Use-Cases werden als Ellipsen und Akteure als Strichmännchen dargestellt. Die Symbole können mit Namen spezifiziert werden. Die Beziehung zwischen Use-Case und Akteur wird mit ‘communicates’ bezeichnet.

Use-Cases können andere Use-Cases benötigen. In Abbildung 29 sieht man diese Beziehungen zwischen Use-Cases. Hier unterscheidet man ‘uses’ und ‘extends’. Eine uses-Beziehung von einem Use-Case A zu einem Use-Case B besagt, daß eine Instanz eines Use-Case A eine Instanz eines Use-Cases B verwendet. Eine extends-Beziehung von einem Use-Case A zu einem Use-Case B besagt, daß bei einer Instanziierung eines Use-Case B das von Use-Case A spezifizierte Verhalten erweitert wird. Die grafische Notation ist entgegengesetzt zu der uses-Beziehung.

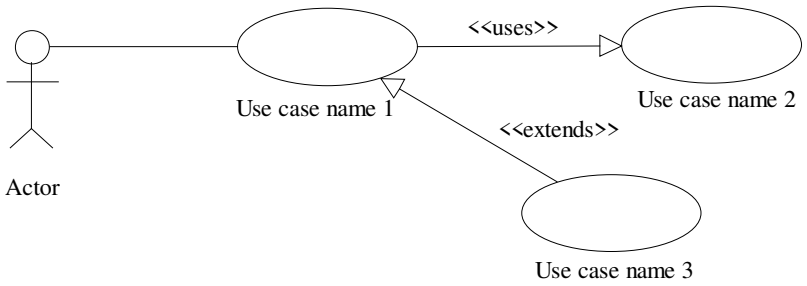


Abbildung 29: Use-Cases

2.2.5 Sequenzdiagramm

Die Sequenzdiagramme veranschaulichen den Nachrichtenaustausch zwischen Objekten mit Berücksichtigung der Zeit. Verwendung finden sie hauptsächlich bei der Modellierung von Echtzeit-Systemen und komplexen Szenarien.

Ein Sequenzdiagramm besteht aus senkrechten gestrichelten Linien, die die Lebenslinien von Objekten darstellen und gerichteten Kanten zwischen diesen senkrechten Linien, die die Nachrichten zwischen den Objekten darstellen. Die Lifelines sind am oberen Ende mit dem Objektnamen beschriftet. Der Nachrichtenname steht an der gerichteten Kante. Die Zeit verläuft bei diesen Diagrammen von oben nach unten. Beziehungen zwischen Objekten werden nicht dargestellt.

Die Aktivität von Objekten wird durch schmale Rechtecke auf den Lifelines dargestellt. Die Art der Aktivität geht aus der Nachricht hervor oder kann als Text neben das Rechteck geschrieben werden. Rekursion wird dadurch dargestellt, daß ein weiteres Rechteck neben das andere gezeichnet wird. Das explizite Löschen eines Objekts wird durch ein ‘X’ ausgedrückt. Dieses steht entweder an der Nachricht, die die Zerstörung des Objekts verursacht oder bei Selbstzerstörung am Ende des Rechteckes.

Objekte können Nachrichten an sich selbst schicken. Dann führt die Nachrichten-Kante zu dem sendenden Objekt zurück. Die Beschriftung besteht aus einem Namen und ggf. Argumenten. Eine Sequenznummer und Bedingungen können optional angegeben werden. Im Normalfall wird eine Nachrichtenkante horizontal gezeichnet. Soll einer Nachricht explizit eine Zeitdauer zugeschrieben werden, so kann der Startpunkt der Kante höher als der Endpunkt der Kante plaziert werden. Weitere Details werden in [UML 1.1 Notation-Guide] aufgeführt. Abbildung 30 zeigt ein Beispiel für ein Sequenzdiagramm.

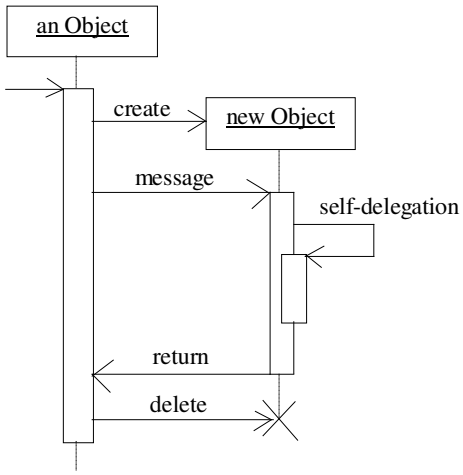


Abbildung 30: Sequenzdiagramm

2.2.6 Kollaborationsdiagramm

Kollaborationsdiagramme stellen die Interaktion von Objekten mittels Nachrichten und die Beziehungen der Objekte zueinander in einem Diagramm dar. Eine grafische zeitliche Dimension, wie bei den Sequenzdiagrammen, gibt es nicht. Zeitliche Aspekte können nur durch Sequenznummern in den Nachrichten dargestellt werden. In der Abbildung 31 sind dies die Nummern 1, 1.1 und 1.2. Nebenläufige Nachrichten sind in diesem Beispiel die Nachrichten 1.2 und 1.2.

Eine Kollaboration besteht aus Objekten und Verbindungen zwischen diesen. Dargestellt wird sie durch ein Kollaborationsdiagramm ohne Nachrichten. Dieses bildet den Kontext für Interaktionen, das dynamische Verhalten. Eine Interaktion besteht aus einer Menge von Nachrichten. Zu einer Kollaboration kann es viele Interaktionen geben. Eine Kollaboration bezieht sich auf einen Use-Case oder eine Operation. Nicht dargestellte Objekte werden in dem entsprechenden Use-Case bzw. der Operation nicht benötigt.

Werden Objekte während der Interaktion neu erzeugt, so werden sie mit '{new}' gekennzeichnet, werden sie vernichtet mit '{destroyed}'. Werden sie erzeugt und vernichtet, bekommen sie die Kennzeichnung '{transient}'.

Den Nachrichten werden Pfeile hinzugefügt, die die Richtung des Nachrichtenflusses angeben. Eine weitere Ergänzung sind die Sequenznummern, die die Reihenfolge der Nachrichten festlegen. Eine Verbindungsbeschriftung hat die Form: `predecessor guard-condition sequence-expression return-value:=message-name argument-list`.

Abbildung 31 zeigt ein Kollaborationsdiagramm mit synchroner Kommunikation, d.h., daß zu einer gesendeten Nachricht auf eine Antwort gewartet wird. Die Möglichkeit der asynchronen Kommunikation ist rechts oben in der Abbildung dargestellt.

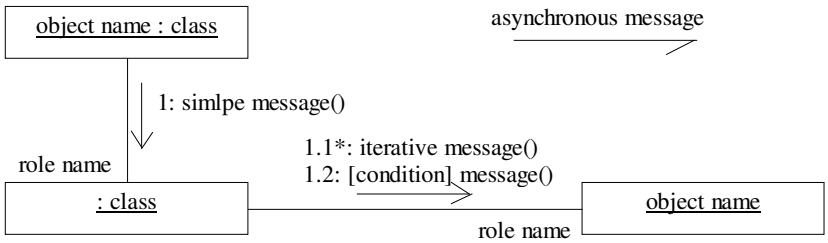


Abbildung 31: Kollaborationsdiagramm

Wird nicht ein bestimmtes Objekt adressiert, sondern eine Menge von Objekten, so kann ein Multiobjekt definiert werden. Die Notation hierfür besteht aus zwei versetzt gezeichneten Rechtecken. Die Abbildung 32 zeigt ein Multiobjekt. Eine Nachricht an das Multiobjekt liefert eine Referenz auf ein bestimmtes Objekt.

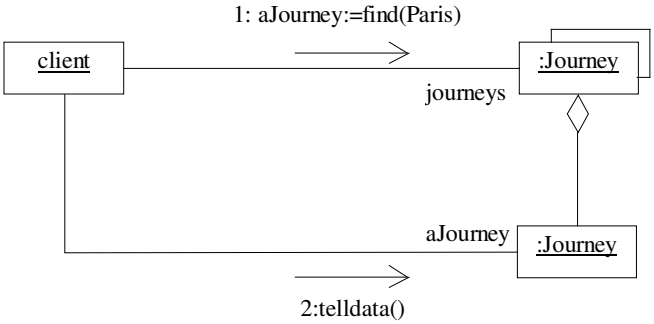


Abbildung 32: Multiobjekt

2.2.7 Zustandsdiagramm

Ein Zustandsdiagramm zeigt die verschiedenen möglichen Zustände eines Objekts, die Zustandsübergänge und die Reaktion des Objekts auf Ereignisse. Die Zustände werden als Rechtecke mit abgerundeten Ecken und die Zustandsübergänge als gerichtete Kanten zwischen den Symbolen dargestellt.

Ein Objekt kann in einem Zustand Aktionen ausführen oder auf Ereignisse warten. Ereignisse überführen ein Objekt von einem Zustand in einen anderen. Aktionen sind atomare Handlungen, d.h. sie sind ununterbrechbar. Einem Zustand können entry- und exit-Aktionen zugeordnet werden. Diese Aktionen werden bei jedem Eintreten bzw. Verlassen des Zustandes ausgeführt. Der Zustandsname ist optional. Weitere Aktionen, die den Zustand nicht ändern, werden im Zustandssymbol textuell notiert. Aktionen sind Reaktionen auf Ereignisse.

Die Notation von Ereignissen ist:
`event-name argument-list[guard-condition]/action-expression`

Bei den entry- bzw. exit-Aktionen ist die Notation:
`entry/action-expression` bzw. `exit/action-expression`

Das keyword '`do`' in der Verwendung '`do/state-machine-name(argument-list)`' weist auf geschachtelte Zustandsdiagramme hin. Dieses Unterzustandsdiagramm muß einen Anfangs- und einen Endzustand haben. Abbildung 33 zeigt ein Zustandsdiagramm mit Unterzuständen.

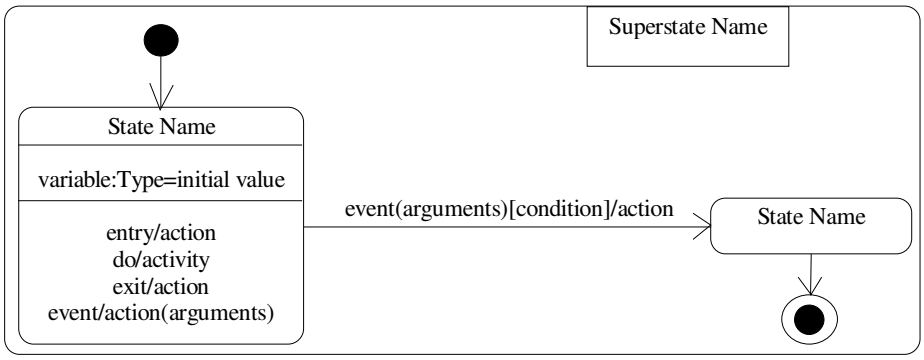


Abbildung 33: Geschachteltes Zustandsdiagramm

Zustände können verfeinert dargestellt werden. Diese Dekomposition kann aus einer Oder-Beziehung oder einer Und-Beziehung bestehen. Bei der Oder-Beziehung handelt es sich um Unterzustände, die sich gegenseitig ausschließen und bei der Und-Beziehung um parallele Unterzustände. Nebenläufigkeit ist in Abbildung 34 dargestellt. Der Zustand ist durch eine gestrichelte Linie in Regionen eingeteilt, in denen sich die nebenläufigen Zustandsdiagramme befinden.

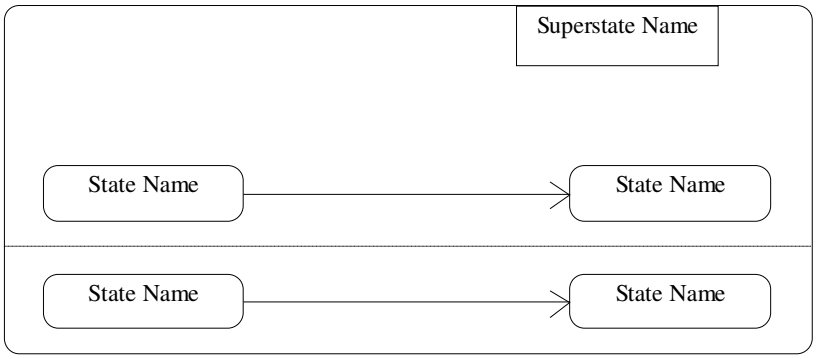


Abbildung 34: Nebenläufigkeit

2.2.8 Komponentendiagramm

Diese Art Diagramme zeigt die Abhängigkeiten der Softwaremodule untereinander. Softwaremodule werden als Komponenten bezeichnet. Komponenten können zu verschiedenen, sich nicht ausschließenden, Zeiten existieren: zur Compile-Zeit, zur Link-Zeit oder zur Laufzeit. Ein Diagramm besteht aus Rechtecken mit je zwei kleinen Rechtecken an einer Seite, den Komponenten, und gestrichelten Linien, den Abhängigkeiten zwischen ihnen. Abbildung 35 zeigt diese Notation.

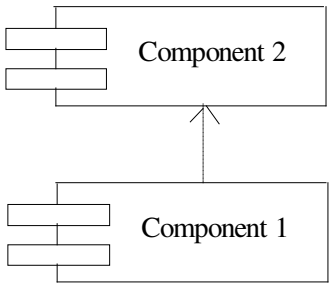


Abbildung 35: Komponenten

2.2.9 Notiz, Einschränkung, Kommentar und abgeleitete Elemente

Notizen werden in Diagrammen verwendet, um zusätzliche Information zu den Diagrammelementen zu integrieren. Die Abbildung 36 zeigt ein Beispiel. Enthalten sein können Bilder, Kommentare, Code o.a.

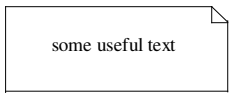


Abbildung 36: Notiz

Einschränkungen definieren bestimmte Bedingungen, die erfüllt werden müssen, um die Gültigkeit des Modells bei der Ausführung zu bewahren. Man unterscheidet vordefinierte Einschränkungen und benutzerdefinierte. Eine Einschränkung wird in ‘{}’ geschrieben. Einschränkungen sind Texte, die direkt an Modellelemente oder in Notizen geschrieben werden. Abbildung 37 zeigt eine Einschrän-

kung in einer Notiz.

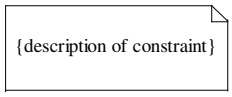


Abbildung 37: Constraint

Im Unterschied zu Einschränkungen sind Kommentare nur für Menschen lesbare Information, nicht im Modell ausführbare. Kommentare werden auch in Notizen geschrieben. Die Klammern werden weggelassen, um sie von Constraints zu unterscheiden. Kommentare und Einschränkungen können mit den betreffenden Elementen durch eine gestrichelte Linie verbunden sein.

Ein abgeleiteter Wert wird von einem oder mehreren anderen Element(en) abgeleitet. Ein Derived element wird mit einem vorangestellten '/' gekennzeichnet. Als Beispiel betrachte man Abbildung 38.

Abbildung 38: Abgeleitetes Element

2.2.10 Vorgehen bei der Analyse

Ein Vorgehen wird in der Definition der Unified Modeling Language nicht vorgeschlagen, um die Modellierungssprache universell für verschiedene Vorgehensweisen einsetzen zu können. UML besteht 'nur' aus einer Menge von wohldefinierten Notationen. Die Entscheidung über das Vorgehen bei der Modellierung liegt beim Entwickler. Hier werden kurz einige Ansätze verschiedener Autoren vorgestellt.

Die Autoren von UML schlagen den iterativen und inkrementellen Rational Objectory Process zum Einsatz von UML zur Analyse und zum Design vor. Strukturiert ist er unter den Gesichtspunkten der Zeit und der modellierten Prozesse. Eine Übersicht über den Rational Objectory Process bietet Quatrani in [Quatrani 98] und die Internetseite 'www.rational.com'.

Die objektorientierte Systementwicklung in [Oesterreich 97] folgt einem evolutionären, iterativen Vorgehensmodell. Dort werden die Entwicklungsaktivitäten in kleine Einheiten gegliedert und die Aktivitäten der unterschiedlichen Detaillierungsebenen in Bezug zueinander koordiniert. Die Entwicklung vom Groben zum Detail erfolgt nicht synchron für alle Bereiche des Systems, sondern situativ je nach Priorität und Problemstellung.

Der in [Kahlbrandt 98] vorgestellte Entwicklungsprozeß ist Use-Case-orientiert, architekturzentriert, iterativ und inkrementell. Betrachtet wird ein iterativer Prozeß über die Phasen Vorstudie, Analyse, Design, Realisierung, Abnahme und Einführung. Die Analyse beginnt mit der Definition der Use-Cases (Anwendungsfälle), mit denen die Leistung des Systems aus der Sicht von Systembenutzern beschrieben wird. Ein Use-Case entspricht einer Menge von Szenarien der Systemnutzung. Die Szenarien werden in Interaktionsdiagrammen dargestellt, in denen Objekte benötigt werden. Durch die Definition dieser Objekte werden Objektklassen gefunden. Diese Klassen und deren Beziehungen

zueinander werden in Klassendiagrammen dargestellt. Bei der Entwicklung der Klassendiagramme steht eine gute Systemarchitektur im Fokus der Modellierung. Zustandsdiagramme und Aktivitätsdiagramme werden zur Darstellung des möglichen dynamischen Verhaltens von Objekten eingesetzt. Die Entwicklung der verschiedenen Diagramme beeinflußt sich gegenseitig. Die Anzahl der Iterationen ist abhängig von der Komplexität des Systems und der Erfahrung der Entwickler.

Larman stellt in [Larman 98] keinen neuen oder standardisierten Entwicklungsprozeß vor, sondern faßt unter dem Namen RPM (Recommended Process and Models) Aspekte bestehender Entwicklungsansätze zusammen. Der effektive Einsatz der Unified Modeling Language wird im wesentlichen durch die Erfahrung des Entwicklers geprägt. Abhängig von der Art des zu entwickelnden Systems können unterschiedliche Aspekte bei der Modellierung im Vordergrund stehen. Empfohlen wird ein iteratives, inkrementelles und Use-Case-orientiertes Vorgehen. Als Grundlage dienen die Ansätze ‘Fusion-Method’ aus [Coleman 94], ‘Martin-Odell’ aus [Martin & Odell 95], ‘Responsibility-Driven Design’ aus [Wirfs-Brock et al. 90], ‘OOSE (Objectory) ’ von [Jacobson 92], ‘OMT’ aus [Rumbaugh et al. 91] und die ‘Booch-Method’ aus [Booch 94].

3 SPEZIFIKATION VON MINITOURS MIT OMT UND UML

In diesem Kapitel wird die Systemanalyse für das Beispielreiseunternehmen ‘Minitours’ durchgeführt. Ausgehend von der textuellen Systembeschreibung werden die von OMT bzw. UML vorgeschlagenen Techniken zur Modellierung verwendet. Die Techniken werden in einer iterativen Vorgehensweise eingesetzt. Wegen der Übersichtlichkeit des Fallbeispiels wird kein spezielles, aus der oben erwähnten Literatur bekanntes, Vorgehen angewendet.

3.1 Minitours

Bei Minitours handelt es sich um eine vereinfachte Variante des Reiseunternehmens Minotours aus [Netzebandt 97]. Statt Flugreisen werden Busreisen angeboten und Buchungen von Häusern sind zunächst nicht vorgesehen. Der Name ‘Minitours’ weist darauf hin, daß nur sehr einfache Busreisen angeboten werden. Dieses Reiseunternehmen ist ein idealisiertes Busreiseunternehmen und in Anlehnung an reale Systeme von mir entwickelt. Erfahrungen mit der Buchung von Reisen und Telefonate mit Reiseunternehmen, sowie die Betrachtung von Fahrplänen haben die Beschreibung des Beispielsystems beeinflußt. Nach einer ausführlichen Systembeschreibung werden in diesem Abschnitt einige Einschränkungen gegenüber realen Systemen erläutert.

3.1.1 Systembeschreibung

Bei dem Reiseunternehmen handelt es sich um ein Busreiseunternehmen, das auf die Basisfunktionen eines solchen Systems reduziert ist. Das Busreiseunternehmen Minitours bietet dem Kunden die Möglichkeit, sich über das Reiseangebot zu informieren, eine Busreise zu buchen und bei der Abfahrt entsprechend betreut zu werden.

Die möglichen Ziele und Reisepreise sind in einer Broschüre zusammengestellt, die dem Kunden als Informationsgrundlage dient. In dieser Broschüre sind alle vom Reiseunternehmen angebotenen Reiseschemata aufgelistet. Für die einzelnen Reiseschemata finden sich Informationen über Abreiseort, Abreisewochentag, Abreiseuhrzeit, Reisepreis pro Person, Ankunftsort, Ankunftswochentag und Ankunftsuhrzeit. Der Kunde kann diese Broschüre durchsehen und sich anhand der Angaben die für ihn relevante Reise heraussuchen.

Bei der Buchung wird davon ausgegangen, daß sich der Kunde über die möglichen Busreisen informiert hat und dem Mitarbeiter nun die zu buchenden Reisedaten nennt. Die benötigten Daten sind Abreiseort, Abreisedatum, Abreiseuhrzeit und Ankunftsort. Der Mitarbeiter überprüft, ob noch ein Sitzplatz in dem zur gewünschten Reise eingesetzten Reisebus verfügbar ist. Ist dies der Fall, so nennt der Mitarbeiter dem Kunden den Reisepreis und fragt ihn, ob er buchen möchte.

Bei einer Buchung reserviert der Mitarbeiter den Sitzplatz. Sofern noch keine Kundendaten aufgenommen wurden, erfragt er die Personalien des Kunden. Die benötigten Daten sind Nach- und Vorname, ggf. Straße, Hausnummer, Postleitzahl, Wohnort und Telefonnummer. Sind die Kundendaten dem Reiseunternehmen bereits bekannt, so genügt i.a. die Angabe des Nach- und Vornamens.

Nach dem Registrieren des gebuchten Sitzplatzes und dem Vermerk der Busreise für den Kunden wird das Reiseticket erstellt und ausgedruckt. Die Angaben auf dem Ticket sind Personentitel, Nachname, Buchungsnummer, Abreiseort, Abreisedatum, Abreiseuhrzeit, Ankunftsort, Ankunftsdatum, Ankunftsuhrzeit, Reisepreis und Buchungsdatum. Nachdem der Kunde die Angaben auf dem Ticket überprüft hat, bezahlt er den Reisepreis und erhält das Ticket.

Zum Abschluß des Buchungsvorgangs vermerkt der Mitarbeiter die Zahlung bezüglich dieses Kunden. Für jeden Kunden, der mindestens eine Reise gebucht hat, gibt es genau einen Vermerk mit den folgenden Angaben: Personentitel, Nachname, Vorname, Straße, Hausnummer, Postleitzahl, Wohnort, Telefonnummer und eine Liste mit den von ihm gebuchten Reisen. Für jede gebuchte Reise werden folgende Daten festgehalten: Buchungsnummer, Abreiseort, Abreisedatum, Abreiseuhrzeit, Ankunftsort, Ankunftsdatum, Ankunftsuhrzeit, Reisepreis und Buchungsdatum.

Unmittelbar vor dem Reiseantritt muß der Kunde einem Minitours-Mitarbeiter das Ticket vorlegen. Nach der Entwertung des Tickets erhält der Kunde eine Bordkarte, die ihn zum Betreten des Reisebusses berechtigt. Die Bordkarte ist mit einer Nummer beschriftet, die den richtigen Bus angibt. Außer der Bordkarte erhält der Kunde Gepäckanhänger, die mit der dem Ticket entsprechenden Buchungsnummer und dem Zielort beschriftet sind. Ein Reisender erhält maximal zwei Gepäckanhänger.

3.1.2 Einschränkungen

Die Funktionalität des Busreiseunternehmens Minitours wird in einigen Punkten, im Vergleich zu existierenden Reiseunternehmen, eingeschränkt. Diese Einschränkungen wurden vorgenommen, um die Komplexität der Systemanalyse zu begrenzen. Eine Analyse eines vollständig detaillierten Reiseunternehmens ist nicht das Ziel dieser Arbeit. Die folgenden Punkte werden bei Minitours nicht berücksichtigt:

- Buchungen für mehr als eine Person
- Umbuchungen
- Ermäßigungen für Studenten, Rentner u.ä.
- Stornierungen
- Unterscheidung von Hin- und Hin-& Rücktickets
- Kombination verschiedener Reisen

3.2 Spezifikation von Minitours mit OMT

Zuerst wird ein Objektmodell entwickelt. Anhand dieser Entwicklung wird das Erlernen der objektorientierten Vorgehensweise dokumentiert. Bei der Entwicklung der verschiedenen Objektmodelle gibt es mehrere Iterationen. Die verschiedenen Modellierungsentscheidungen bei diesem iterativen Prozeß werden in diesem Kapitel dokumentiert. Zu dem Objektmodell wird dann ein dynamisches und ein funktionales Modell erstellt.

3.2.1 Objektmodell

Das Objektmodell besteht aus Objektklassen und Beziehungen zwischen ihnen. Zur Darstellung wird ein Objektdiagramm verwendet. Dieses Objektdiagramm wird hier schrittweise entwickelt. Bei dem ersten Entwurf werden alle Schritte sehr ausführlich dokumentiert. Das erhaltene Objektmodell wird dann diskutiert. Bei den darauf folgenden Iterationen wird jeweils ein neues Objektdiagramm vorgestellt und die wesentlichen Veränderungen werden erläutert. Die einzelnen Schritte bei den Iterationen werden aus Redundanzgründen nicht so ausführlich beschrieben wie beim Entwurf des ersten Modells.

3.2.1.1 Identifikation der Objektklassen

Anhand der Systembeschreibung werden Objekte bzw. Objektklassen identifiziert und aufgeschrieben. Objekte werden nur im Singular notiert. Bei der ersten Analyse sind dies:

- Reiseunternehmen, Buchung, Busreise, Ziel, Reisepreis, Broschüre, Kunde, Anfragedaten, Mitarbeiter, Reiseziel, Sitzplatz, Bus, Kundendaten, Ticket, Geld, Leistung, Beratung, Abfertigung, Stadt, Abreiseort, Reisepreis, Ankunftsort, Reisevorschlag, Bordkarte, Gepäckanhänger, Gepäckstücke

Aus unterschiedlichen Gründen kann es sein, daß einige der gefundenen Klassen nicht benötigt werden. Die folgende Liste enthält Kriterien zum Auffinden dieser Klassen und die entsprechenden Klassen des Minitoursbeispiels:

- Überflüssige Klassen
  - Reiseunternehmen: Die Menge aller Objekte, bis auf den Kunden, stellt das Reiseunternehmen dar.
  - Abfertigung: Die Abfertigung wird zunächst nicht modelliert.
  - Bordkarte: Bordkarten sind Teil der Abfertigung.
  - Gepäckanhänger: Gepäckanhänger sind Teil der Abfertigung.
- Irrelevante Klassen
  - Beratung: Der Kunde informiert sich in der Broschüre oder fragt nach konkreten Daten. Die Frage nach einer konkreten Reise ist Teil des Gesprächs mit dem Mitarbeiter. Ein Gespräch wird hier nicht als Objekt gesehen.
  - Stadt: Städte treten nur als Attribute der Busreisen auf und sind als eigene Objekte irrelevant.
  - Gepäckstücke: Gepäckstücke werden erst bei der Abfertigung berücksichtigt, und dort auch nur deren Anzahl.
- Wage Klassen
  - Leistung: Die Leistung des Reiseunternehmens wird nicht als Objekt betrachtet. Die Leistung wird durch eine Teilmenge der Objekte realisiert.
- Attribute
  - Abreiseort: Attribut einer Anfrage und einer Busreise.
  - Ziel=Reiseziel=Ankunftsort: Das Ziel ist das Attribut einer Anfrage und einer Busreise.
  - Reisepreis: Der Reisepreis ist ein Attribut einer Busreise.
  - Anfragedaten: Die Anfragedaten sind Teil des Informationsgesprächs.
  - Reisevorschlag: Der Reisevorschlag ist Teil des Informationsgesprächs.
  - Sitzplatz: Attribut einer Busreise.
  - Bus: Attribut einer Busreise.
- Operationen
  - Buchung: Die Buchung ist ein Vorgang, bei dem Daten geändert werden.

3.2.1.2 Anfertigung eines Datenwörterbuchs

Das Datenwörterbuch enthält die Beschreibung aller Objektklassen bezüglich der Anwendung . Es können schon hier Attribute und Operationen, sowie Assoziationen beschrieben werden. Im Laufe dieses Kapitels wird noch ausführlich auf diese Aspekte eingegangen.
Hier noch einmal die identifizierten Objektklassen der ersten Analyse von Minitours im Überblick:

- Busreise=Busreisedaten, Broschüre, Busreiseinfo, Kunde, Mitarbeiter, Kundendaten, Ticket, Geld=Zahlung, Datenbank

Das Datenwörterbuch sieht wie folgt aus:

- Broschüre:
  Die Broschüre enthält eine Auflistung aller vom Reiseunternehmen angebotenen Busreisen. Zu jeder Reise gibt es Busreiseinformationen.
- Busreiseinformation (Busreiseinfo):
  Die Informationen zu einer Busreise in der Broschüre bestehen aus: Abreiseort, Abreisewochentag, Abreiseuhrzeit, Reisepreis, Ankunftsort, Ankunftswochentag, Ankunftsuhrzeit, Zwischenstops inkl. Ankunfts- und Abfahrtszeiten. Die Busreiseinformationen sind das Erzeugungsmuster für Busreisen. Bei den Busreiseinformationen werden nur die Wochentage genannt, an denen die Reise stattfindet, kein konkretes Datum.
- Busreisedaten:
  In der Datenbank werden für jede konkrete Busreise Busreisedaten geführt. Diese enthalten folgende Einträge: Abreiseort, Abreisedatum, Abreiseuhrzeit, Busnummer, freie Plätze, belegte Plätze, Ankunftsort, Ankunftsdatum, Ankunftsuhrzeit, Zwischenstopliste inkl. der Ankunfts- und Abfahrtszeit, Fahrerliste, Reisepreis pro Platz. Die Anzahl der freien und belegten Plätze wird bei einer Buchung geändert.
- Datenbank:
  Eine Datenbank wird an dieser Stelle eingeführt, weil dieser Begriff ausdrückt, daß dort alle Daten im Reiseunternehmen gespeichert werden. Die Datenbank enthält zwei Arten von Daten: Busreisedaten und Kundendaten. Diese Daten sind folgendermaßen verknüpft: Wird bei einer Buchung in den Kundendaten eine Reise eingetragen, so wird bei den zugehörenden Busreisedaten die Anzahl der freien Plätze um 1 erniedrigt und die Anzahl der belegten Plätze um 1 erhöht. Sind keine freien Plätze vorhanden, so kann die Busreise nicht gebucht werden. Wenn noch keine Kundendaten existieren, werden sie angelegt.
- Kunde:
  Bei einer Buchung werden von dem Kunden folgende Daten benötigt: Name, Vorname, Straße, Hausnummer, Postleitzahl, Wohnort, Telefonnummer.
- Kundendaten:
  Die Kundendaten enthalten die persönlichen Daten eines Kunden und die reisespezifischen Daten der gebuchten Reise(n). Die persönlichen Daten sind: Titel, Name, Vorname, Straße, Hausnummer, Postleitzahl, Wohnort, Telefonnummer. Die reisespezifischen Daten bestehen aus einer Liste von folgenden Einträgen für jede gebuchte Reise: Buchungsnummer, Abreiseort, Abreisedatum, Abreiseuhrzeit, Ankunftsort, Ankunftsdatum, Ankunftsuhrzeit, Zwischenstopliste inkl. der Ankunfts- und Abfahrtszeit, Reisepreis pro Platz, Buchungsdatum. Ändern sich die Kundendaten, so werden sie aktualisiert.
- Mitarbeiter:
  Über den Mitarbeiter sind folgende Daten bekannt: Name, Vorname, Straße, Hausnummer, Postleitzahl, Wohnort, Telefonnummer.
- Ticket:
  Auf dem gedruckten Ticket finden sich folgende Informationen: Titel, Name, Buchungsnummer, Abreiseort, Abreisedatum, Abreiseuhrzeit, Ankunftsort, Ankunftsdatum, Ankunftsuhrzeit, Zwischenstopliste inkl. der Ankunfts- und Abfahrtszeit, Reisepreis, Buchungsdatum. Das Ticket wird auf einem Drucker ausgedruckt.

- Zahlung:
  Eine Zahlung entspricht der Höhe des zu zahlenden Reisepreises und kann auf verschiedene Arten geleistet werden: Bar, per Scheck oder per Kreditkarte. Die Währung der Zahlung muß der Währung des Landes entsprechen, in dem die Zahlung vorgenommen wird. Die Art der Zahlung wird bei der Buchung festgelegt.

3.2.1.3 Finden der Assoziationen

Assoziationen entsprechen den Verben oder Verbphrasen in der Systembeschreibung. Die Assoziationen des ersten Minitoursmodells können dem Objektdiagramm in Abbildung 39 entnommen werden.

Es kann vorkommen, daß unnötige oder falsche Assoziationen gefunden wurden. Das bisher entwikkelte Objektmodell wird bezüglich der Kriterien in [Rumbaugh et al. 91] untersucht. In diesem Modell werden alle Assoziationen so gelassen, wie sie in Abbildung 39 definiert sind. Die Assoziationen werden im Diskussionsteil dieses Abschnitts besprochen.

3.2.1.4 Identifizieren der Attribute

Aus der Systembeschreibung werden die benötigten Attribute für die Objektklassen gewonnen. Attribute, die nicht explizit genannt werden, müssen in Gesprächen mit dem Auftraggeber und aus Erfahrung ermittelt werden. Die folgende Liste enthält alle Objekte mit ihren Attributen:

- Broschüre:
  Besteht aus Busreiseinformation.
- Busreiseinformation:
  Abreiseort, Abreisewochentag, Abreiseuhrzeit, Reisepreis, Ankunftsort, Ankunftswochentag, Ankunftsuhrzeit, Zwischenstops inkl. Ankunfts- und Abfahrtszeiten
- Busreisedaten:
  Abreiseort, Abreisedatum, Abreiseuhrzeit, Busnummer, freie Plätze, belegte Plätze, Ankunftsort, Ankunftsdatum, Ankunftsuhrzeit, Zwischenstopliste inkl. der Ankunfts- und Abfahrtszeit, Fahrerliste, Reisepreis pro Platz
- Datenbank:
  Ist Aggregat aus Kundendaten und Busreisedaten.
- Kunde:
  Name, Vorname, Straße, Hausnummer, Postleitzahl, Wohnort, Telefonnummer
- Kundendaten:
  Titel, Name, Vorname, Straße, Hausnummer, Postleitzahl, Wohnort, Telefonnummer, Liste für jede gebuchte Reise (Buchungsnummer, Abreiseort, Abreisedatum, Abreiseuhrzeit, Ankunftsort, Ankunftsdatum, Ankunftsuhrzeit, Zwischenstopliste, Reisepreis pro Platz, Buchungsdatum)
- Mitarbeiter:
  Name, Vorname, Straße, Hausnummer, Postleitzahl, Wohnort, Telefonnummer
- Ticket:
  Titel, Name, Buchungsnummer, Abreiseort, Abreisedatum, Abreiseuhrzeit, Ankunftsort, Ankunftsdatum, Ankunftsuhrzeit, Zwischenstopliste inkl. der Ankunfts- und Abfahrtszeit, Reisepreis, Buchungsdatum
- Zahlung:
  Betrag, Art

3.2.1.5 Testen der Ausführungspfade und Iterationen des Entwicklungsprozesses

In diesem Schritt wird getestet, ob alle benötigten Assoziationen vorhanden sind. Hier werden auch die Multiplizitäten festgelegt. Gewählt wurde hier aufgrund besserer Entwicklung mit dem verwendeten Zeichenprogramm die numerische Darstellung und nicht die symbolische. Die Abbildung 39 zeigt den ersten Ansatz für ein Objektmodell.

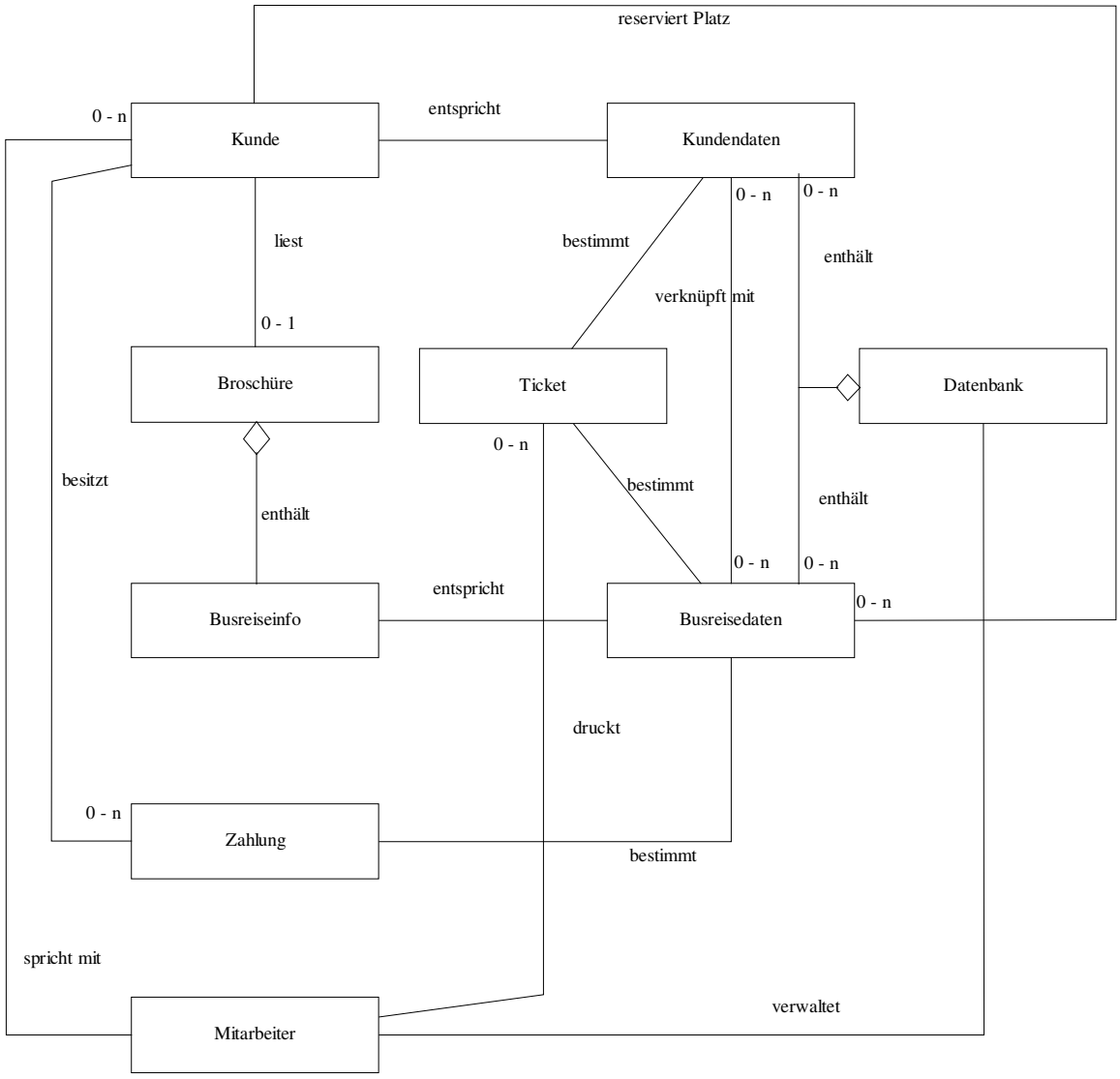


Abbildung 39: Erster Vorschlag für das Objektmodell

Es folgt eine Erläuterung des in Abbildung 39 dargestellten Objektmodells. Die Diskussion erfolgt bezüglich der Elemente des Objektdiagramms, also den Objektklassen, Assoziationen und Multiplizitäten.

Objektklassen:
Die Klassen ‘Kunde’ und ‘Mitarbeiter’ entsprechen den realen Objekten der textuellen Systembeschreibung. Werden diese im System dargestellt, so handelt es sich üblicherweise nur um die im System relevanten Daten dieser Objekte. Im Falle des Kunden ist das die Klasse ‘Kundendaten’. Daß es sich bei dieser Klasse u.a. um die besagten Daten handelt, ist der Sinn dieser Klasse und deshalb ist der Zusatz ‘...daten’ im Klassennamen unnötig. Die Klassen ‘Kunde’ und ‘Mitarbeiter’ sollten durch

Klassen ersetzt werden, die die Funktionalität der Objekte ‘Kunde’ und ‘Mitarbeiter’ widerspiegeln. Die Klasse ‘Datenbank’ ist zu implementationsnah und sollte allgemeiner bezeichnet werden.

Assoziationen:
Die Assoziationsnamen ‘enthält’ sind überflüssig, da die verwendete Aggregation diesen Sachverhalt ausdrückt. Die Namen ‘entspricht’ und ‘bestimmt’ sind überflüssig, weil die Assoziationen und die Angabe von Multiplizitäten diese Art Beziehung zwischen Klassen durch die Existenz schon ausdrücken. Sie wurden hier gewählt, um jede Assoziation zu betiteln.

Multiplizitäten:
Die ‘zu eins’-Beziehungen (unbeschriftete Assoziationsenden) sind an vielen Stellen zu streng. Z.B. muß es in diesem Modell zu einem Mitarbeiter eine Datenbank geben und zu einer Datenbank genau einen Mitarbeiter. Ein anderes Beispiel ist die Assoziation zwischen Busreisedaten und Ticket.

Vererbung:
Vererbung wurde in diesem Modell nicht verwendet, obwohl sie z.B. möglich wäre, indem ‘Kunde’ und ‘Mitarbeiter’ gemeinsam von einer Klasse ‘Person’ erben.

Die Erkenntnisse aus der Entwicklung dieses ersten Modells führen zu einem zweiten Ansatz zur Modellierung des Objektmodells. Der wesentliche Unterschied zum ersten Modell ist der Versuch, Handlungen als Klassen darzustellen und die zu implementationsnahen Klassen zu eliminieren. Die Abbildung 40 zeigt das Objektdiagramm zum zweiten Ansatz.

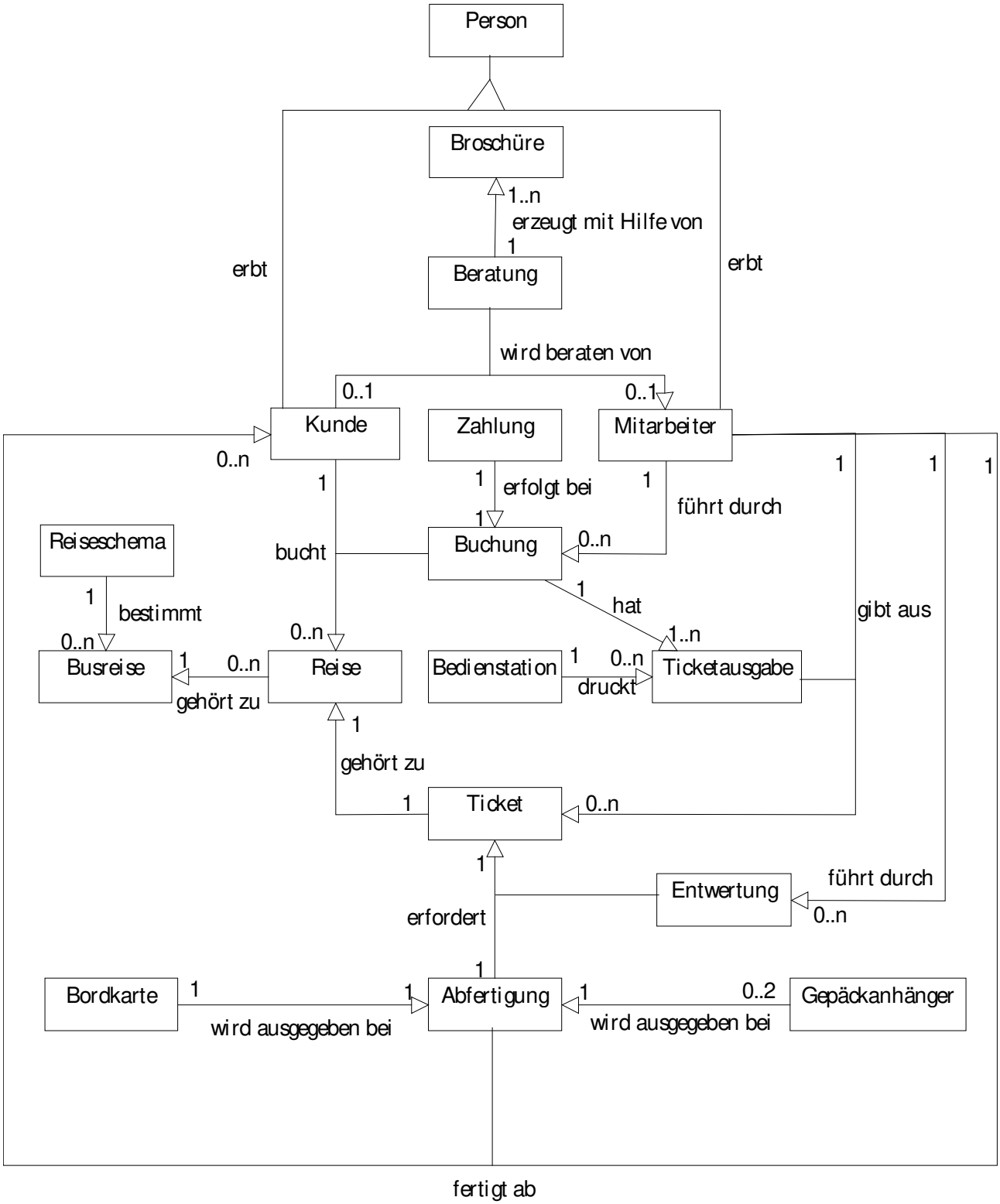


Abbildung 40: Zweiter Vorschlag für das Objektmodell

Objektklassen:
Die in diesem Modell eingeführten Klassen ‘Beratung’, ‘Buchung’, ‘Ticketausgabe’, ‘Entwertung’ und ‘Abfertigung’ entsprechen Handlungen (Systemfunktionen) und sind als Assoziationsklassen modelliert worden. Das Diagramm ist sehr ablauforientiert. Die Klasse ‘Datenbank’ ist in diesem Modell nicht mehr vorhanden. Aus den genannten Gründen ist dieses Diagramm nicht objektorien-

tiert und wird hier nur aufgeführt, um zu zeigen, wie die Technik der Objektdiagramme unsauber verwendet werden kann.

Assoziationen:
Die Assoziationen sind gerichtet und es werden die o.g. Assoziationsklassen verwendet. Die eingeführte Richtung ist oftmals nicht eindeutig und in dieser Verwendung unnötig. Zum Beispiel wird ein Gepäckanhänger bei einer Abfertigung ausgegeben. Genauso hätte die Richtung umgekehrt werden können, mit der Bedeutung, daß eine Abfertigung einen Gepäckanhänger erzeugt und ausgibt. Wie erwähnt liegt hier keine saubere Objektorientierung vor. Die Richtung von Assoziationen wird bei der Codeerzeugung aus Klassendiagrammen interessant.

Multiplizitäten:
Bei einigen Muß-Beziehungen ist zu überlegen, ob diese so nicht zu streng modelliert wurden.

Vererbung:
Die Klassen ‘Kunde’ und ‘Mitarbeiter’ erben in diesem Modell von der Klasse ‘Person’.

Bei dem dritten Ansatz stehen die Daten im Vordergrund, nicht die Handlungen. Die Abbildung 41 zeigt das Objektdiagramm. Die Multiplizitäten wurden etwas lockerer definiert als in den Modellen zuvor. Eine zentrale Position nimmt der Kunde ein.

Objektklassen:
In diesem Modell wird die Klasse ‘Kunde’ als Repräsentation des realen Kunden im System betrachtet. Über diesen sind der Name und die Adresse interessant. Der Mitarbeiter als Klasse wurde eliminiert, da die Informationen über ihn bzgl. des Modellierungsziels nicht relevant sind. Die Attribute wurden explizit modelliert. Auffällig ist die zentrale Position der Kundenklasse.

Assoziationen:
Es wurde versucht, allen Assoziationen Namen zu geben. Dieses erweist sich oftmals als schwierig, besonders bei Beziehungen wie ‘gehört zu’, ‘wird bezahlt an’ oder ‘besitzt’. Hier könnte sich eher die Verwendung von Aggregation eignen.

Multiplizitäten:
Die in diesem Modell meist explizit notierten Multiplizitäten sind teilweise zu streng modelliert. Die Muß-Beziehungen einiger Klassen zum Kunden sind solche Fälle.

Vererbung:
Die gemeinsamen Attribute einiger Klassen und deren ähnlicher Klassenname weisen auf Vererbungsbeziehungen hin. Eingesetzt wurde dieses Konzept hier jedoch nicht.

Im Gegensatz zum dritten Modell spielt im vierten Ansatz die Autorisierung eine zentrale Rolle. Die Abbildung 42 zeigt das Objektdiagramm zu dem vierten Ansatz.

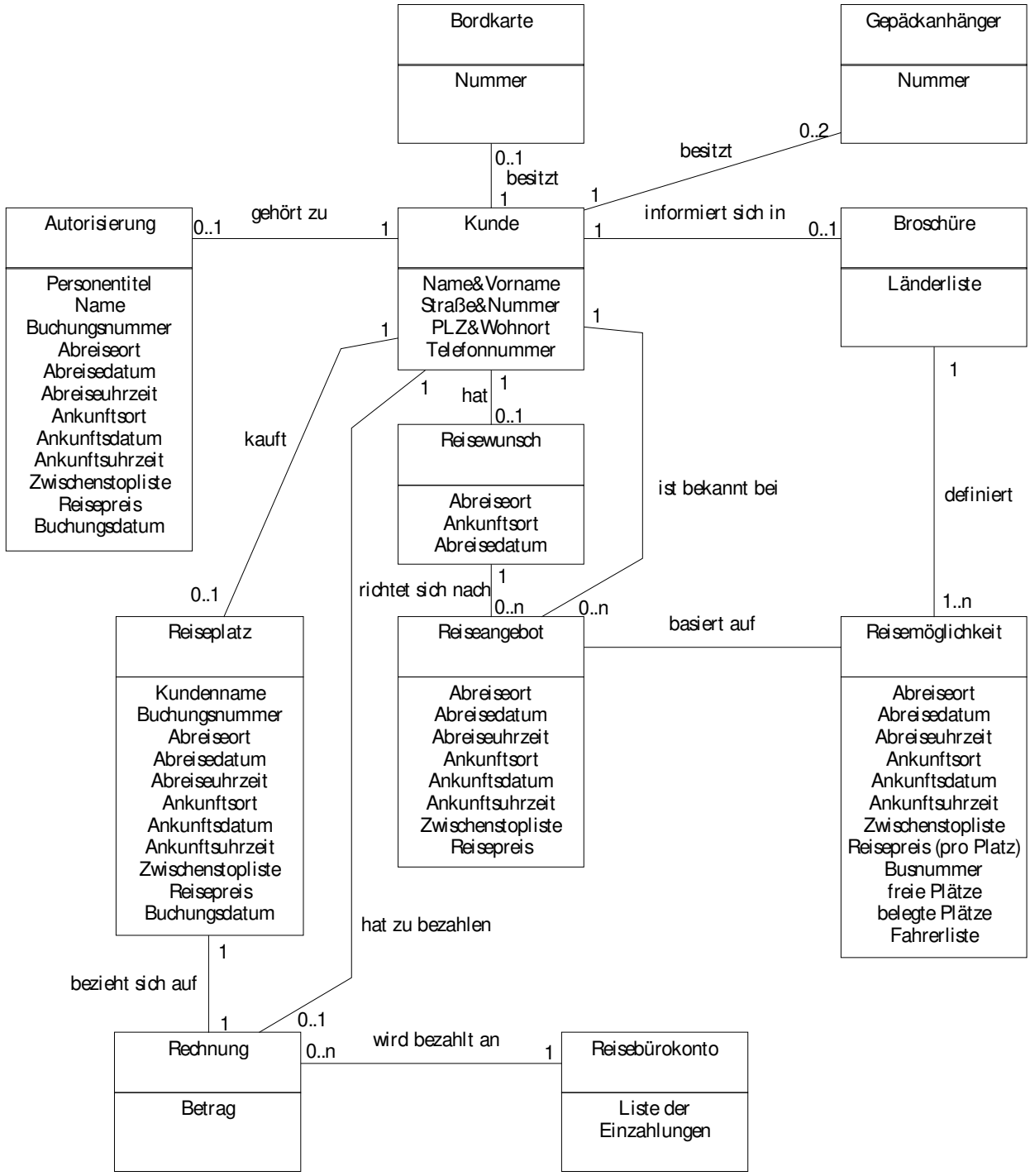


Abbildung 41: Dritter Ansatz

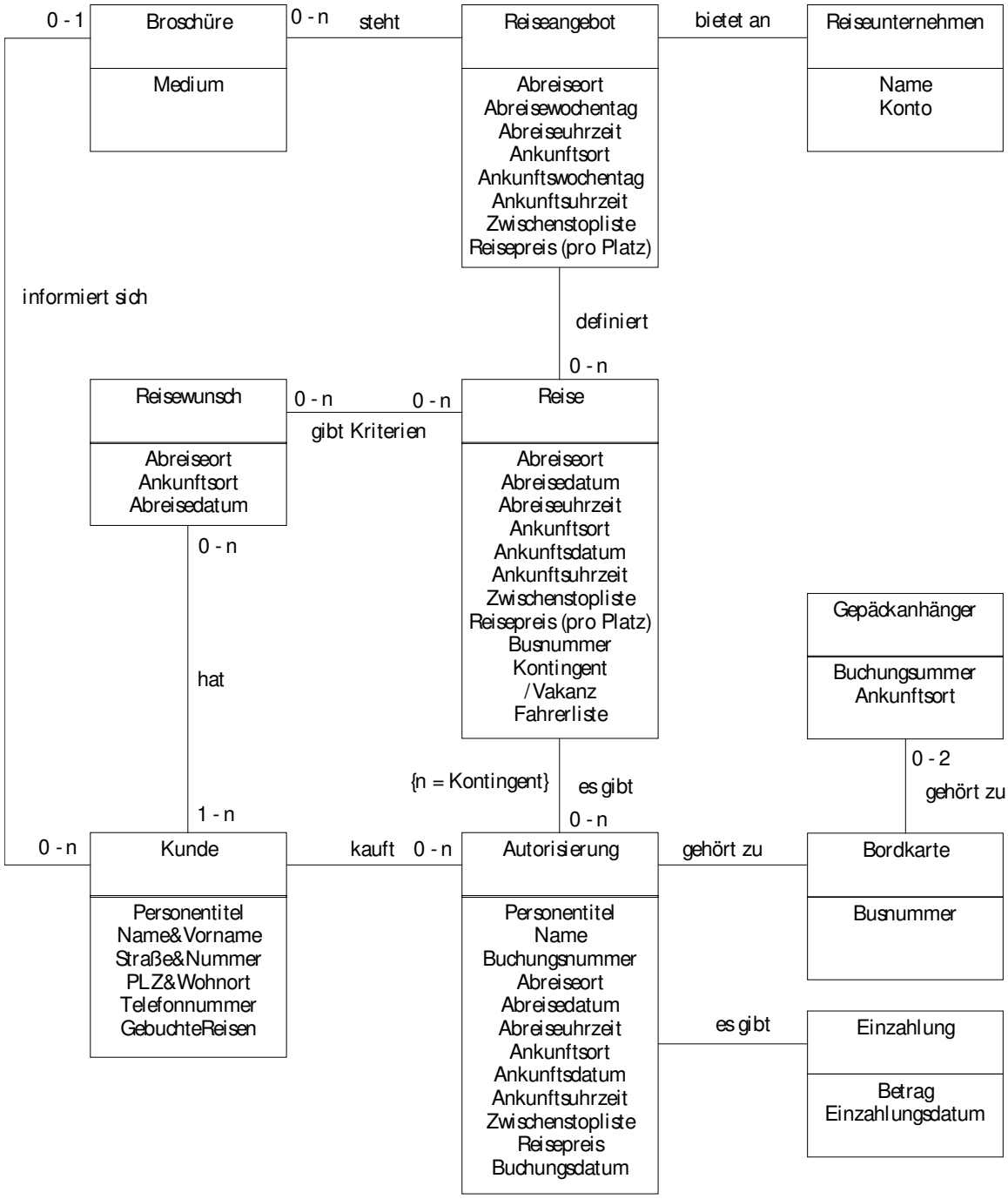


Abbildung 42: Vierter Ansatz

Objektklassen:
In diesem Modell hat die Autorisierung eine zentrale Rolle. Im Gegensatz zum vorigen Modell haben hier Objekte der Klasse ‘Bordkarte’ eine Muß-Beziehung zu der Autorisierung. Ein Gepäckanhänger gehört zu genau einer Bordkarte. Auch die Einzahlung hat eine Beziehung zu der Autorisierung. In diesem Modell kann ein Kunde viele Autorisierungen erwerben, im vorigen Modell nur eine. Das Reisebürokonto ist als Attribut der neu eingeführten Klasse ‘Reiseunternehmen’ modelliert worden, da ein Konto hier nicht als Objekt relevant ist. Ein Objekt der Klasse ‘Reiseunternehmen’ übernimmt die Rolle des Objekts, das ein Reiseangebot anbietet.

Assoziationen und Multiplizitäten:
Die Namen der Assoziationen ‘gibt Kriterien’, ‘es gibt’ und ‘gehört zu’ sind nicht gut gewählt. Bes-

ser wäre es, diese durch andere Notationen auszudrücken. Hierzu könnten Kommentare oder Aggregation verwendet werden. Die Muß-Beziehungen sind teilweise zu streng. Zu einer Autorisierung muß es ja nicht immer eine Bordkarte geben, nur irgendwann kann es mal eine geben. Auf der anderen Seite muß eine Bordkarte zu genau einer Autorisierung gehören.

Abbildung 43 zeigt das endgültige Objektmodell, das Teil des Analysemodells ist. Es besteht aus den Klassen inklusive ihrer Attribute, den Assoziationen zwischen den Klassen und den Multiplizitäten.

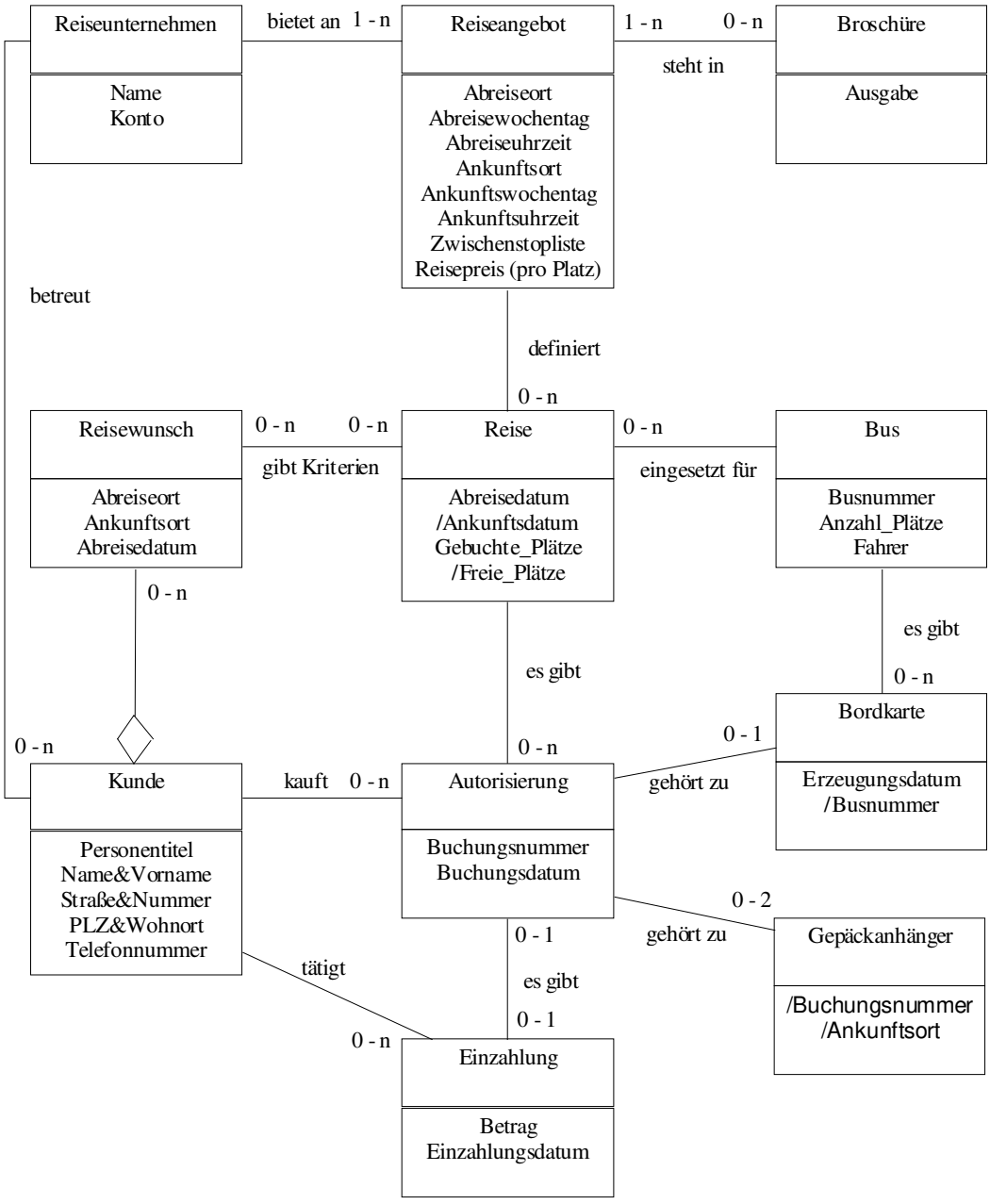


Abbildung 43: Objektmodell

Das Datenwörterbuch enthält eine Beschreibung aller Klassen mit ihren Attributen und Assoziationen. Verfaßt ist es in natürlicher Sprache.

Datenwörterbuch:

- Autorisierung
  Eine Autorisierung wird für einen Platz einer Reise bei der Buchung an einen Kunden ausgegeben und berechtigt diesen, an der Reise teilzunehmen. Eine Autorisierung ist durch eine Buchungsnummer gekennzeichnet und enthält das Buchungsdatum. Sie ist nicht übertragbar, d.h. sie ist nur von dem Kunden benutzbar, dessen Name bei der Buchung angegeben wurde.
- Bordkarte
  Zu jeder entwerteten Autorisierung wird eine Bordkarte ausgegeben, die den richtigen Bus anhand seiner Nummer identifiziert und den Kunden zum Betreten des Busses berechtigt. Zu einem Bus werden maximal soviele Bordkarten ausgegeben, wie es Plätze im Bus gibt. Eine Bordkarte ist mit dem Datum deren Erzeugung versehen.
- Broschüre
  Wenn das Reiseunternehmen eine Broschüre erstellt, so enthält diese alle Reiseangebote. Zusätzlich enthält sie eine Angabe über die Gültigkeitsdauer im Attribut ‘Ausgabe’.
- Bus
  Busse werden für die angebotenen Reisen eingesetzt. Wichtig für das Reiseunternehmen ist die Anzahl der Plätze, eine zugewiesene Busnummer und welcher Fahrer eingesetzt wird.
- Einzahlung
  Eine Einzahlung wird zu einer Buchung, d.h. zu einer Autorisierung, vom Kunden getätigt. Die Reisen werden sofort bei der Buchung bar bezahlt.
- Gepäckanhänger
  Gepäckanhänger werden, wie auch die Bordkarte, bei der Abreise bzw. Entwertung der Autorisierung ausgegeben. Ein Kunde darf maximal zwei Gepäckstücke mitnehmen, d.h. er erhält maximal zwei Gepäckanhänger.
- Kunde
  Über einen Kunden können im Reiseunternehmen bestimmte Daten bekannt sein. Dies sind die Daten zur Identifikation eines Kunden und die zur Korrespondenz notwendigen, d.h. Name, Adresse und Telefonnummer. Interessant ist ein Kunde dadurch, daß er Reisewünsche hat.
- Reise
  Eine Reise hat ein konkretes Abreise- und Ankunftsdatum, sowie gebuchte und freie Plätze. Für eine Reise wird maximal ein Bus eingesetzt. Die Anzahl der freien Plätze berechnet sich aus der Anzahl der zur Verfügung stehenden Plätze im Bus und den schon gebuchten Plätzen.
- Reiseangebot
  Ein Reiseangebot entspricht einem Reiseschema für alle Reisen von einem Abfahrtsort zu einem Ankunftsort an einem bestimmten Wochentag. Weitere Attribute sind die Abfahrts- und Ankunftsuhrzeit, Zwischenstops und der Reisepreis für einen Platz. Ein Reiseangebot definiert mehrere Reisen mit jeweils konkretem Reisedatum. Ein Reiseangebot ist in einer Broschüre enthalten, wenn das Reiseunternehmen eine Broschüre herausgibt.
- Reiseunternehmen
  Das Reiseunternehmen, in diesem Fall ist es Minitours, bietet Reiseangebote an. Attribute des Reiseunternehmens sind der Name, in diesem Fall ist er ‘Minitours’, und ein Konto für die Einzahlungen. Das Reiseunternehmen betreut die Kunden.
- Reisewunsch
  Ein Reisewunsch wird von einem Kunden formuliert und gibt Kriterien an, nach denen Reisen seitens des Reiseunternehmens vorgeschlagen werden. Im wesentlichen sind dies der Abfahrts- und Ankunftsort, meistens auch ein Abreisedatum.

Die Operationen sind hier nicht enthalten, weil sie erst im funktionalen Modell definiert werden. In Bezug auf das hier gezeigte Modell wird auch das dynamische Modell entworfen.

Objektklassen:
In diesem Modell wird eine Klasse ‘Bus’ explizit modelliert, von der ein Objekt für ein Objekt der Klasse ‘Reise’ eingesetzt wird. Die anderen Klassen bleiben unverändert. Ein weiterer Unterschied zu den anderen Ansätzen ist die implizite Darstellung von Attributen. Die zu einer Autorisierung gehörenden Angaben über den Kunden können durch die eindeutige Assoziation zu dem Kunden ermittelt werden. Deshalb werden sie hier nicht explizit im Objektdiagramm dargestellt. Bei den Attributen mit dem vorangestellten Schrägstrich (/) handelt es sich um abgeleitete Attribute. Z.B. wird das Ankunftsdatum bei der Reise aus dem Abreisedatum und der Reisedauer (Differenz zwischen Ankunftswochentag und Abreisewochentag) berechnet. Die freien Plätze berechnen sich aus der Differenz der Anzahl der Plätze im Bus und den schon gebuchten Plätzen. Bei den Klassen ‘Bordkarte’ und ‘Gepäckanhänger’ werden die Attribute explizit erwähnt, um zu zeigen, welche Informationen die Objekte dieser Klasse enthalten.

Assoziationen:
Die Assoziation zwischen ‘Kunde’ und ‘Broschüre’ wurde eliminiert. Eingeführt wurde die Beziehung zwischen ‘Kunde’ und ‘Reiseunternehmen’. Die Klasse ‘Kunde’ aggregiert Reisewünsche. Ein Gepäckanhänger hat jetzt eine Beziehung zu einer Autorisierung und nicht mehr zu einer Bordkarte. Die Assoziation zwischen ‘Kunde’ und ‘Einzahlung’ ist überflüssig, da sie über die Beziehungen zur Autorisierung abgeleitet werden kann, verdeutlicht hier aber, daß ein Kunde keine oder viele Einzahlungen tätigen kann und daß eine Einzahlung von genau einem Kunden getätigt wird.

Multiplizitäten:
Ein Reiseunternehmen bietet mindestens ein Reiseangebot oder viele Reiseangebote an und jedes Reiseangebot wird von genau einem Reiseunternehmen angeboten. Ein Reiseangebot steht in keiner oder vielen Broschüren und eine Broschüre enthält mindestens ein Reiseangebot. Zu einem Reiseangebot gibt es viele Reisen und eine Reise wird von genau einem Reiseangebot definiert. Zu einer Reise wird genau ein Bus eingesetzt und ein Bus kann für keine oder mehr Reisen eingesetzt werden. Zu einem Bus gibt es viele Bordkarten und eine Bordkarte ist für genau einen Bus gültig und gehört zu genau einer Autorisierung. Zu einer Autorisierung gibt es keine oder eine Bordkarte und bis zu zwei Gepäckanhänger, wobei ein Gepäckanhänger zu genau einer Autorisierung gehört. Zu einer Autorisierung gibt es eine Kann-Beziehung zu einer Einzahlung, da diese nicht sofort bei Existenz der Autorisierung getätigt werden muß. Eine Einzahlung wird für maximal eine Autorisierung getätigt.

Das Reiseunternehmen betreut keine oder mehrere Kunden und ein Kunde wird von genau einem Reiseunternehmen betreut. Ein Kunde kann keine oder viele Reisewünsche haben. Diese werden deshalb als Aggregation dargestellt. Eine für einen Kunden relevante Reise richtet sich nach seinem Reisewunsch. Ein Kunde kauft für die gewünschte Reise eine oder viele Autorisierungen und eine Autorisierung wird von genau einem Kunden gekauft. Diese Autorisierung ist für genau eine Reise gültig, wobei es für eine Reise keine oder viele Autorisierungen geben kann.

3.2.2 Dynamisches Modell

Bei der dynamischen Modellierung wird das zeitliche Verhalten des Systems betrachtet. Für jedes Objekt wird ein Zustandsdiagramm angefertigt. Bei der Entwicklung der Zustandsdiagramme werden zuerst Szenarien für die Systemnutzung erstellt. Für diese Szenarien wird zunächst ein Event-Flow-Diagramm entworfen und dann werden Event-Trace-Diagramme entwickelt. Der letzte Schritt ist die Entwicklung der Zustandsdiagramme. Im folgenden wird das dynamische Modell für das Busreiseunternehmen ‘Minitours’ entwickelt.

3.2.2.1 Szenarien für typische Dialoge

Bei der Entwicklung der Szenarien wird das System und ein Systembenutzer betrachtet. Dieses ist bei dem Reiseunternehmen ein externer Kunde. Dieser unterscheidet sich von der Objektklasse ‘Kunde’, die die interne Repräsentation eines Kunden realisiert.

Bei den Szenarien werden im allgemeinen zuerst die Normalfälle beschrieben, dann Ausnahme- und Fehlerfälle. Bei diesem System beginnt immer der Kunde eine Kommunikation mit dem System. Entsprechend [Matthes 97] werden zwischen dem Kunden und dem Dienstanbieter die Phasen Anfrage, Übereinkunft, Leistung und Rückmeldung durchlaufen. Die Normalfälle sind im folgenden aufgeführt. Ein Ausnahmefall bei dem Reiseunternehmen würde dann vorliegen, wenn der Kunde, nachdem er einen Reiseplatz hat buchen lassen, kein Geld hat. Bei dem hier modellierten Reiseunternehmen bezahlt der Kunde bei Erhalten der Autorisierung. Ausnahmefälle werden hier also nicht betrachtet. Fehlerfälle werden erst bei der Implementation betrachtet.

Bei den Szenarien handelt es sich um

- Kunde fragt nach Broschüre
- Kunde informiert sich über die Reiseangebote
- Kunde äußert Reisewunsch; Reise zum Reisewunsch buchbar
- Kunde äußert Reisewunsch; Reise zum Reisewunsch nicht buchbar
- Der Kunde tritt die Reise an

Aus Gründen der besseren Lesbarkeit werden die Szenarien textuell und grafisch zusammen in Abschnitt 3.2.2.4 ausführlich dargestellt.

3.2.2.2 Event-Flow-Diagramm

Das Event-Flow-Diagramm in Abbildung 44 zeigt unter Vernachlässigung der Reihenfolge alle externen Ereignisse aller Szenarien auf einen Blick.

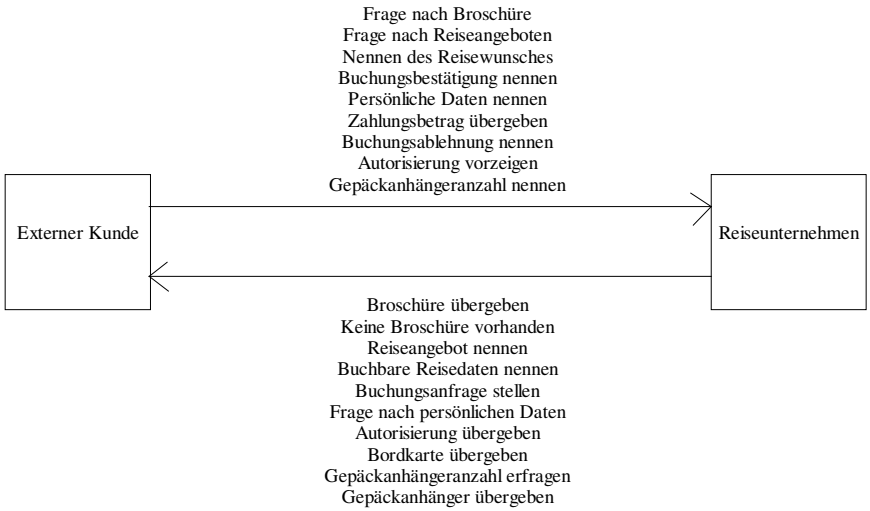


Abbildung 44: Event-Flow-Diagramm für Minitours

3.2.2.3 User-Interface

Bei Minitours könnte für den Systembenutzer ein Mitarbeiter die Benutzerschnittstelle sein. Dieser erfragt den Reisewunsch, also die Daten bzgl. Abfahrtsort, Ankunftsort und evtl. Abfahrtsdatum. Eine Schnittstellenmodellierung besteht aus der Aufnahmemöglichkeit dieser Daten und anderer Dinge seitens des Mitarbeiters. Außerdem Möglichkeiten zur Ausgabe von Daten und Gegenständen.

Interessanter wäre die Schnittstellenmodellierung bei der Kommunikation des Kunden mit einem Buchungscomputer. Hier gibt es sehr viele Möglichkeiten der Dateneinein- und ausgabe, von denen einige skizziert werden könnten. Beispiele sind die Eingabe der benötigten Daten in entsprechende Felder, die Auswahl aus Listen, natürlichsprachliche Eingabe der Daten u.s.w.

3.2.2.4 Identifizieren der Ereignisse

Bei diesem Schritt werden Ereignisse identifiziert, die im System auftreten. Bei der Entwicklung der Event-Trace-Diagramme werden alle Systemobjekte und ein externes Benutzer-Objekt betrachtet. Dieses ist bei dem Reiseunternehmen der externe Kunde. Für jedes Szenario wird ein Event Trace Diagramm entworfen. Dargestellt sind diese Diagramme in den folgenden Abbildungen.

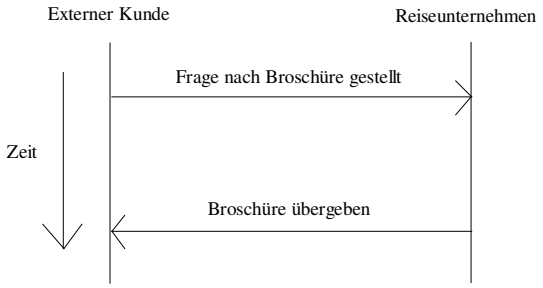


Abbildung 45: Event-Trace-Diagramm zu Szenario 1

- Szenario 1: Kunde fragt nach Broschüre
  - Der Kunde fragt das System nach einer Broschüre.
  - Das System gibt dem Kunden eine Broschüre.

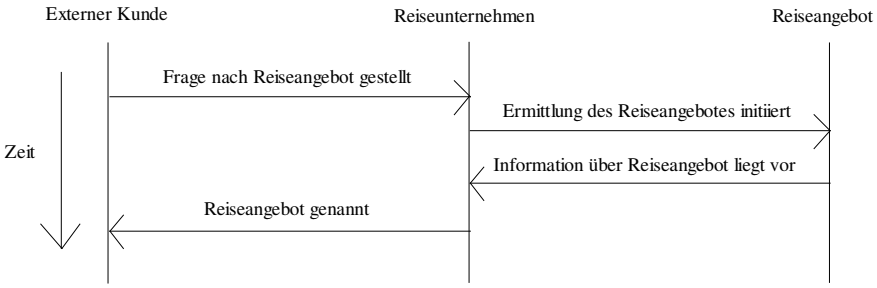


Abbildung 46: Event-Trace-Diagramm zu Szenario 2

- Szenario 2: Kunde informiert sich über die Reiseangebote
  - Der Kunde fragt das System nach Reiseangeboten.
  - Das System ermittelt die Reiseangebote.
  - Das System nennt dem Kunden die Reiseangebote.

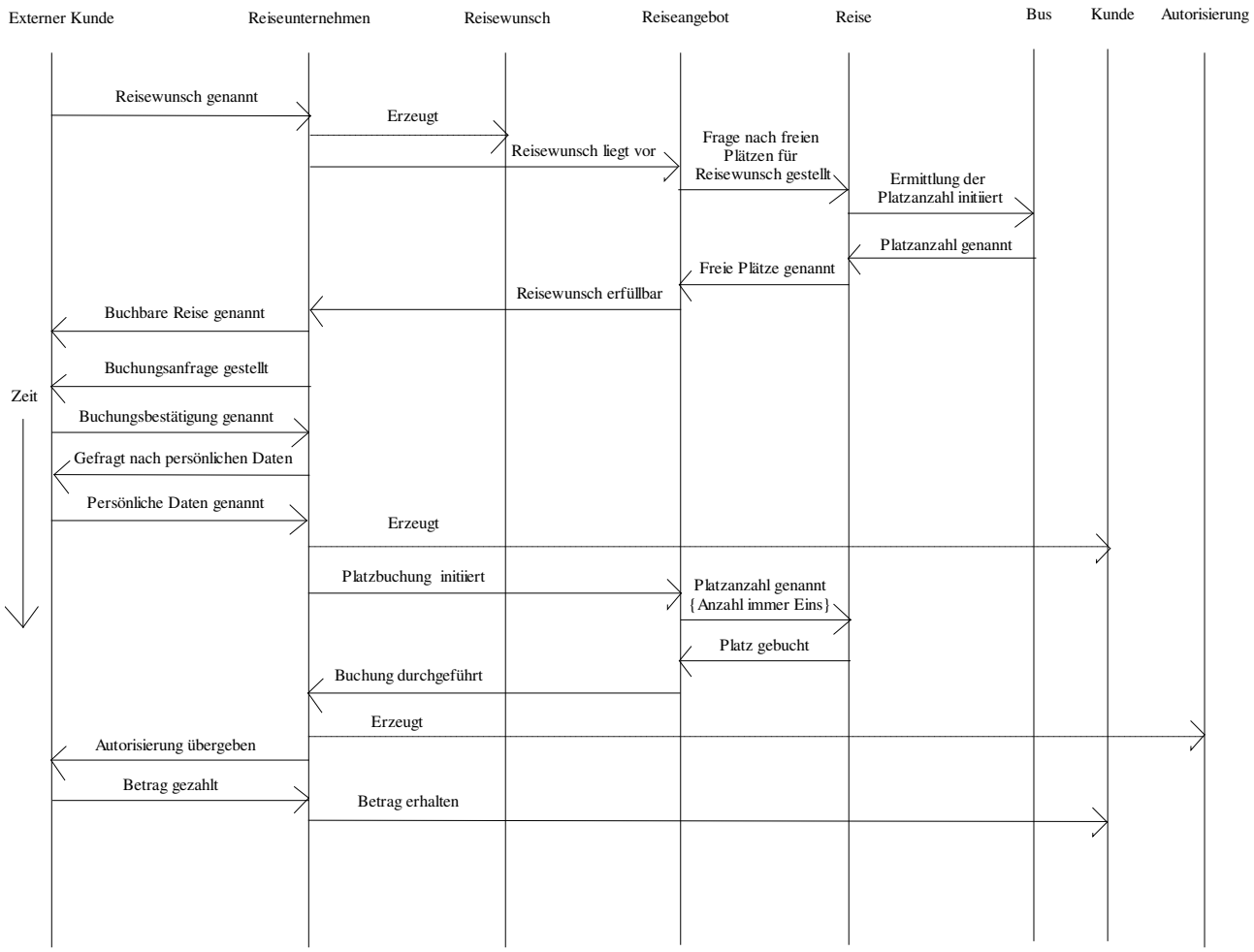


Abbildung 47: Event-Trace-Diagramm zu Szenario 3

- Szenario 3: Kunde äußert Reisewunsch; Reise zum Reisewunsch buchbar
  - Der Kunde richtet an das System einen Reisewunsch.
  - Das System ermittelt die verfügbare(n) Reise(n) zu dem Reisewunsch.
  - Das System nennt dem Kunden die dem Reisewunsch entsprechende(n) buchbare(n) Reise(n).
  - Das System fragt den Kunden, ob er einen Platz einer vorgeschlagenen Reise buchen möchte.
  - Der Kunde bestätigt, daß er buchen möchte.
  - Das System fragt den Kunden nach seinen persönlichen Daten.
  - Der Kunde nennt seine persönlichen Daten.
  - Das System speichert die Kundendaten.
  - Das System bucht einen Platz in der vom Kunden gewählten Reise.
  - Das System speichert die Buchung.
  - Das System erstellt eine Autorisierung.
  - Das System gibt dem Kunden die Autorisierung für die Reise.
  - Der Kunde bezahlt den verlangten Betrag, der dem Reisepreis entspricht.
  - Das System verbucht die Bezahlung.

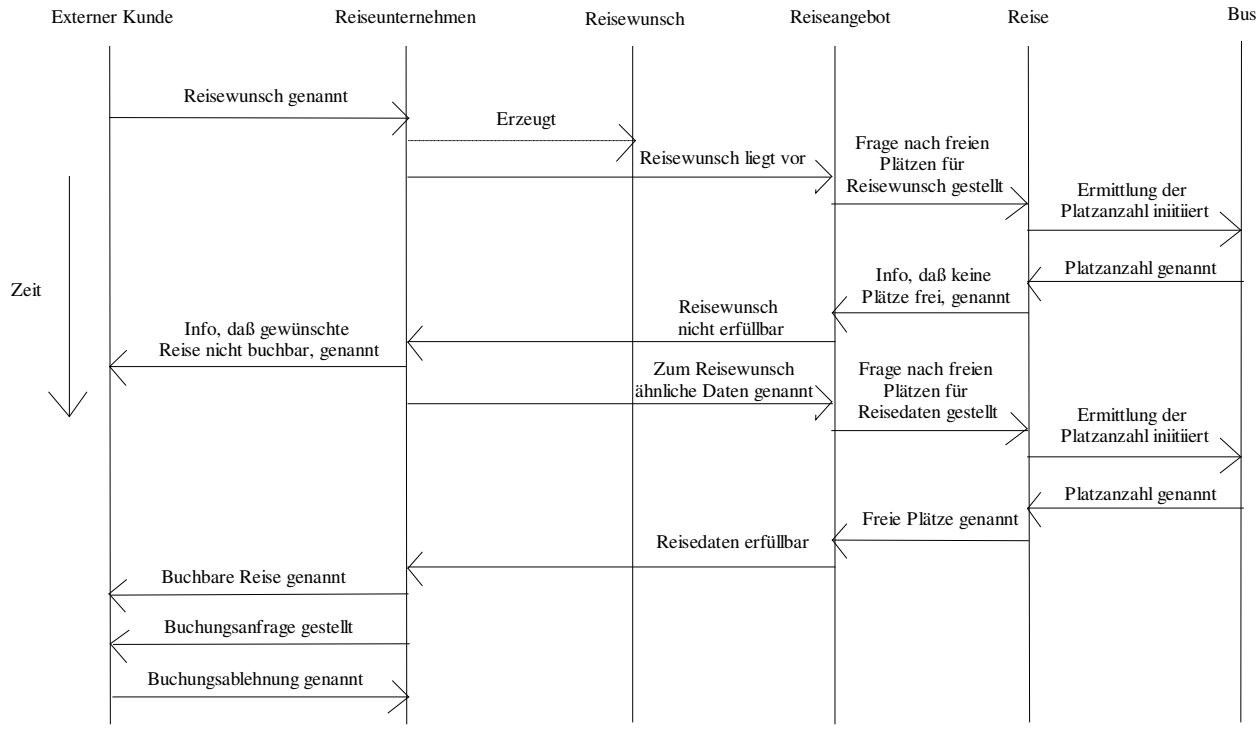


Abbildung 48: Event-Trace-Diagramm zu Szenario 4

- Szenario 4: Kunde äußert Reisewunsch; Reise zum Reisewunsch nicht buchbar
  - Der Kunde richtet einen Reisewunsch an das System.
  - Das System ermittelt die verfügbare(n) Reise(n) zu dem Reisewunsch.
  - Das System informiert den Kunden, daß keine Reise mit den gewünschten Daten buchbar ist.
  - Das System ermittelt ähnliche Reisen, wenn welche buchbar sind.
  - Das System schlägt dem Kunden die Reisen vor.
  - Der Kunde möchte keine der vorgeschlagenen Reisen buchen.

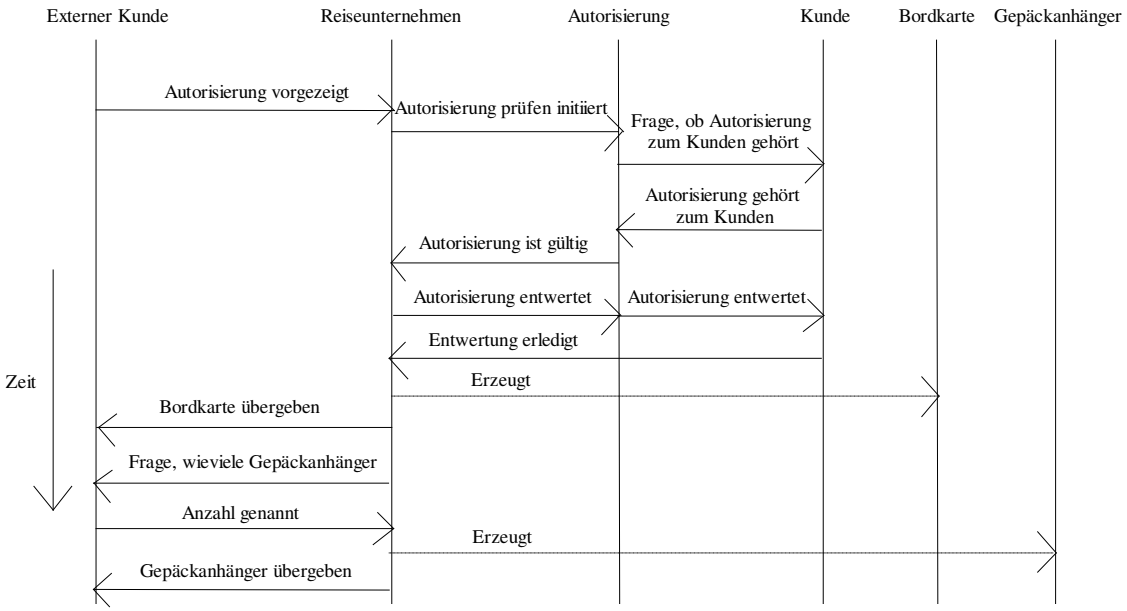


Abbildung 49: Event-Trace-Diagramm zu Szenario 5

- Szenario 5: Der Kunde tritt die Reise an
  - Der Kunde legt die Autorisierung vor.
  - Das System entwertet die Autorisierung und gibt dem Kunden eine Bordkarte.
  - Das System fragt den Kunden, wieviele Gepäckanhänger er benötigt.
  - Der Kunde möchte einen Gepäckanhänger.
  - Das System gibt dem Kunden einen Gepäckanhänger.
  - Der Kunde befestigt den Gepäckanhänger am Gepäck.
  - Der Kunde geht zum Bus, gibt die Bordkarte und das Gepäck ab und steigt in den Bus.

3.2.2.5 Erstellen von Zustandsdiagrammen

Für jede Objektklasse werden die Ereignisse und Zustände in jeweils einem Zustandsdiagramm dargestellt. Diese Zustandsdiagramme stehen über gemeinsame Ereignisse in Verbindung und beschreiben das dynamische Verhalten des Systems. Unterschiedliche Folgezustände eines Zustands können nur durch unterschiedliche Ereignisse erreicht werden. Hier folgen nun die Zustandsdiagramme für die Objektklassen von Minitours.

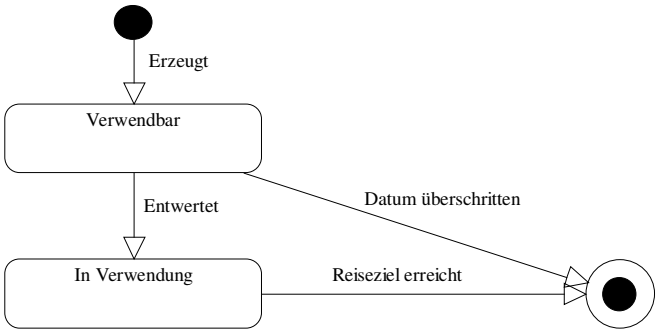


Abbildung 50: Autorisierung

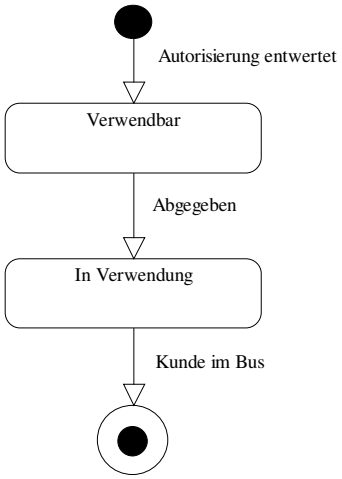


Abbildung 51: Bordkarte

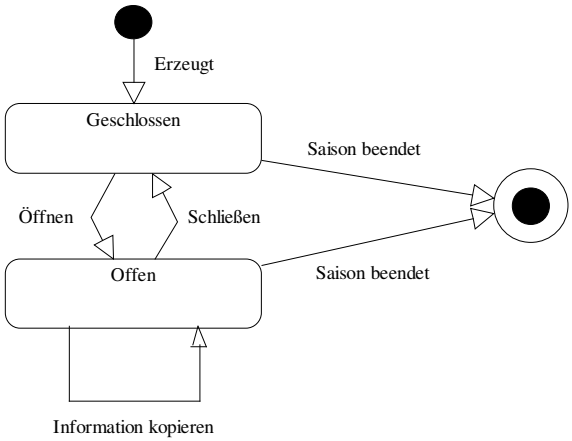


Abbildung 52: Broschüre

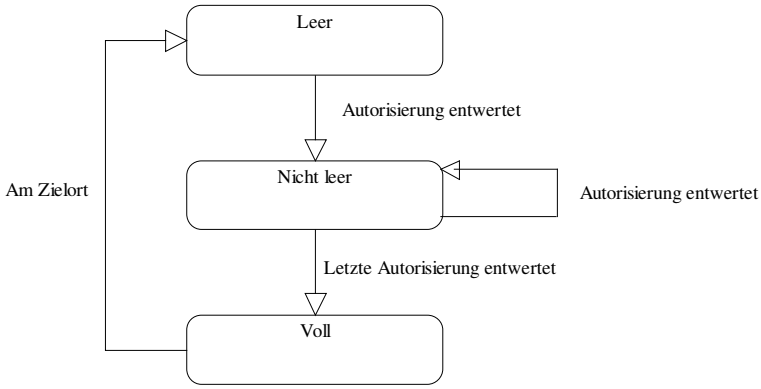


Abbildung 53: Bus

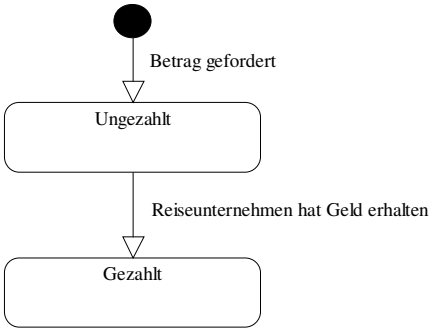


Abbildung 54: Einzahlung

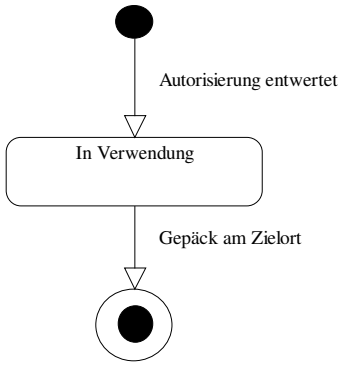


Abbildung 55: Gepäckanhänger

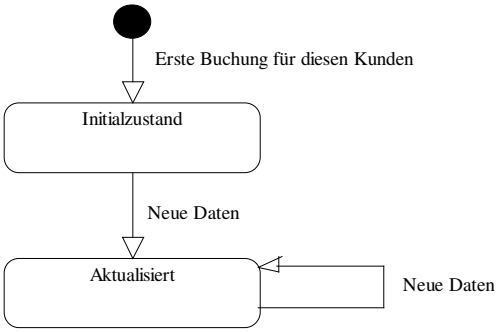


Abbildung 56: Kunde

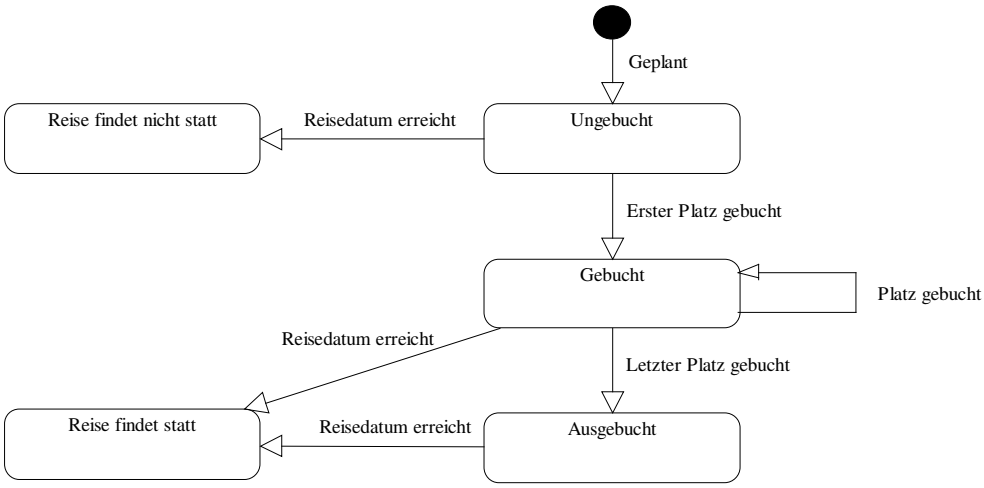


Abbildung 57: Reise

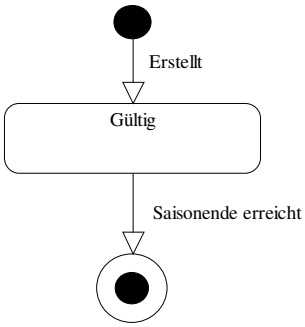


Abbildung 58: Reiseangebot

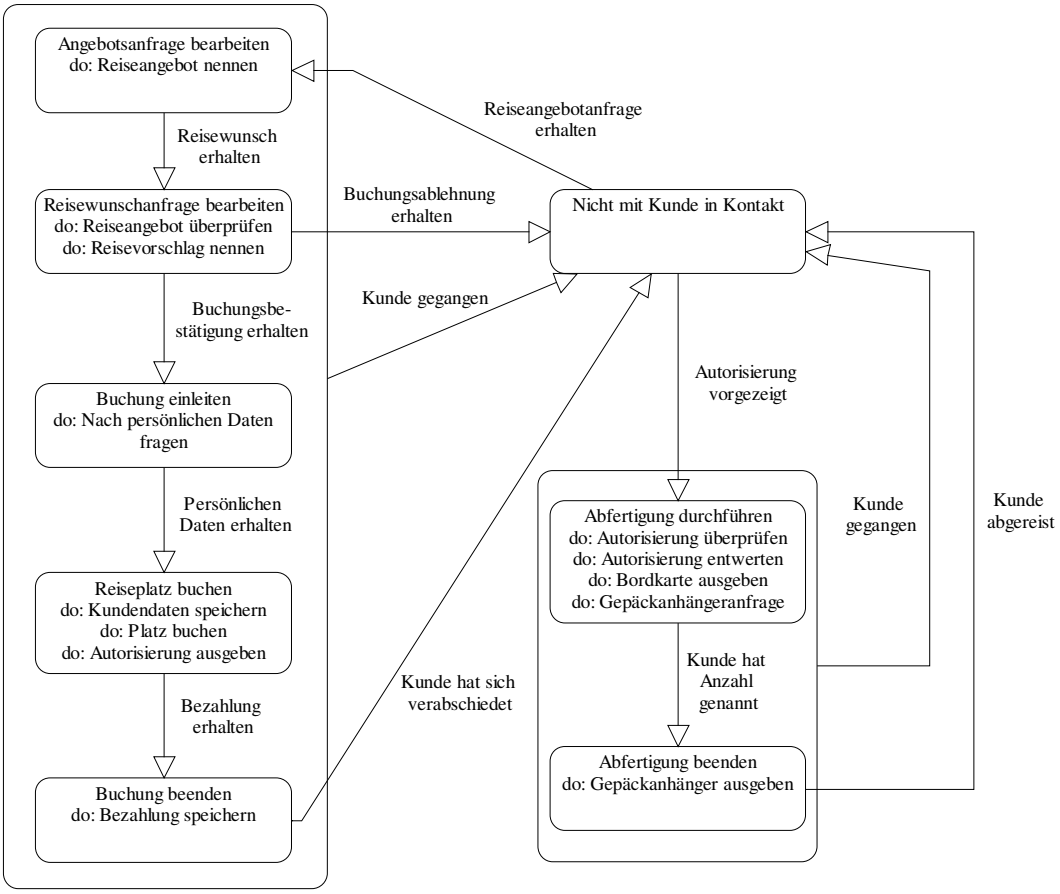


Abbildung 59: Reiseunternehmen

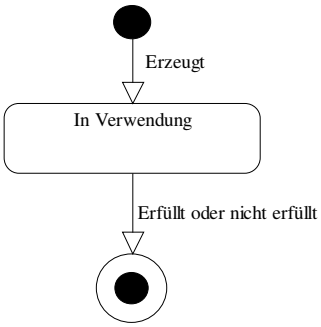


Abbildung 60: Reisewunsch

3.2.3 Funktionales Modell

Bei der funktionalen Modellierung wird anhand von Datenflußdiagrammen beschrieben, wie Werte berechnet werden. Die Prozesse in den Diagrammen entsprechen den Aktivitäten und Aktionen in den Zustandsdiagrammen des dynamischen Modells. Die Datenflüsse entsprechen den Objekten bzw. Attributen des Objektmodells.

3.2.3.1 Identifizieren von Ein- und Ausgabewerten

Ein erstes Diagramm zeigt zunächst, welche Daten zwischen dem System und der Außenwelt ausgetauscht werden. Im Falle des Minitours-Systems besteht die relevante Außenwelt aus dem Kunden. Die Abbildung 61 zeigt den Datenaustausch.

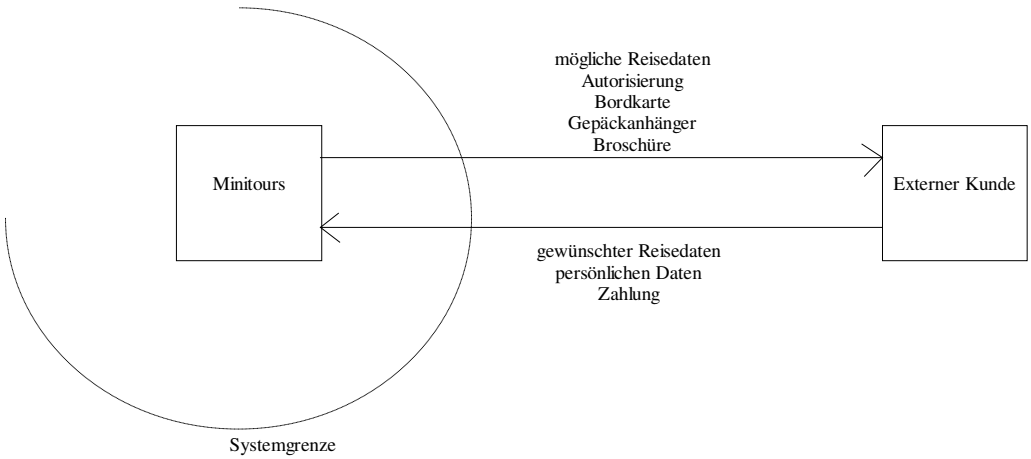


Abbildung 61: Datenein- und Ausgabe

3.2.3.2 Entwicklung von Datenflußdiagrammen

Um zu zeigen, wie die Ausgaben aus den Eingaben berechnet werden, werden Prozesse ausgehend von den Ausgaben entwickelt. Dabei müssen Datenspeicher identifiziert werden. Deshalb muß spätestens an dieser Stelle ein Datenspeicher eingeführt werden. Die Bezeichnung ‘Datenbank’ erscheint zu implementationsnah, beschreibt aber gut den Sachverhalt, daß dort die im System benötigten Daten gespeichert werden. Prozesse können in mehreren Schichten verfeinert dargestellt werden. Die Berechnungsreihenfolge wird hier nicht dargestellt. Dies geschieht im dynamischen Modell.

Der erste Schritt ist die Entwicklung des Top Level Datenflußdiagrammes in Abbildung 62.

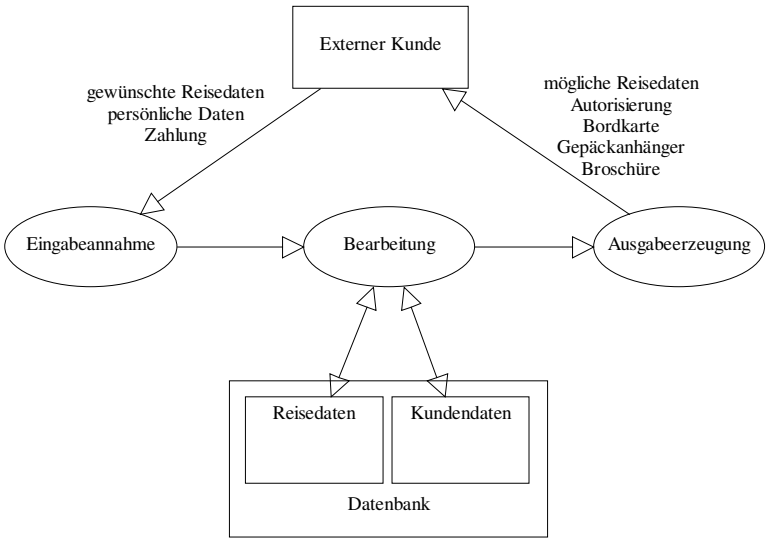


Abbildung 62: Top-Level-Diagramm

Die folgenden Datenflußdiagramme zeigen eine detaillierte Darstellung der Berechnungen. In der Abbildung 63 wird das Aushändigen der Broschüre dargestellt. Eingabe ist die Frage nach einer Broschüre und die Ausgabe ist eine Broschüre. Aus dem Speicher wird eine Broschüre entnommen.

In Abbildung 64 werden die vom Kunden gewünschten Reisedaten auf Verfügbarkeit überprüft. Eingabe sind die gewünschten Reisedaten und Ausgabe sind die möglichen Reisedaten. Aus dem Datenspeicher wird nur gelesen.

In Abbildung 65 werden die persönlichen Daten eines Kunden erfaßt bzw. aktualisiert. Eingabe sind die persönlichen Daten des Kunden, eine Ausgabe gibt es nicht. Auf dem Datenspeicher wird gelesen und ggf. geschrieben.

Abbildung 66 stellt den Prozeß der Platzreservierung da. Benötigte Eingaben sind die möglichen Reisedaten und der Name des Kunden. Auf den Datenspeichern wird gelesen und geschrieben.

Vor der Erstellung einer Autorisierung müssen zuerst, wie in Abbildung 67 dargestellt, die Reisedaten und die Kundendaten ermittelt werden, bevor die Autorisierung erstellt werden kann. Die Erstellung und Ausgabe der Autorisierung wird bei den Kundendaten registriert und die Autorisierung an den Kunden ausgegeben.

Abbildung 68 zeigt den Vorgang einer Zahlung. Diese wird bei den Daten des Kunden und bei den Buchhaltungsdaten des Unternehmens registriert. Die Eingabe ist der Betrag der Zahlung. Eine Ausgabe gibt es hier nicht. Auf den Datenspeichern wird gelesen und geschrieben.

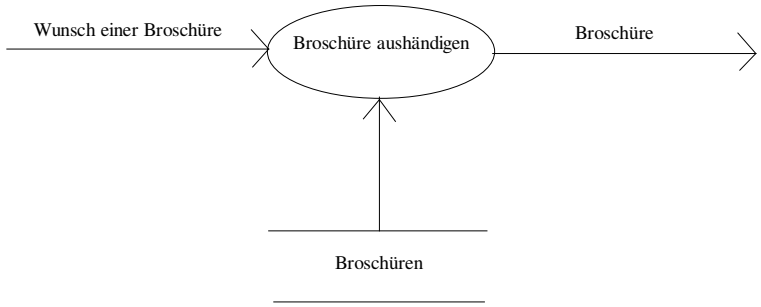


Abbildung 63: DFD Aushändigen der Broschüre

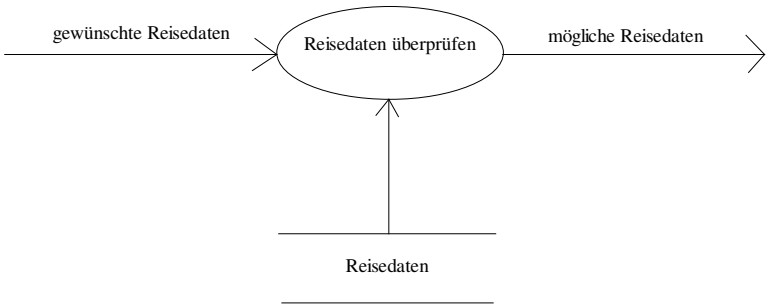


Abbildung 64: DFD Überprüfung der Reisedaten

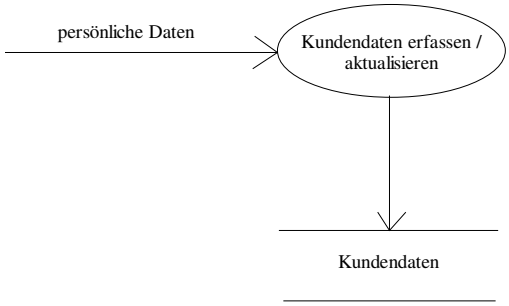


Abbildung 65: DFD Kundendaten erfassen/aktualisieren

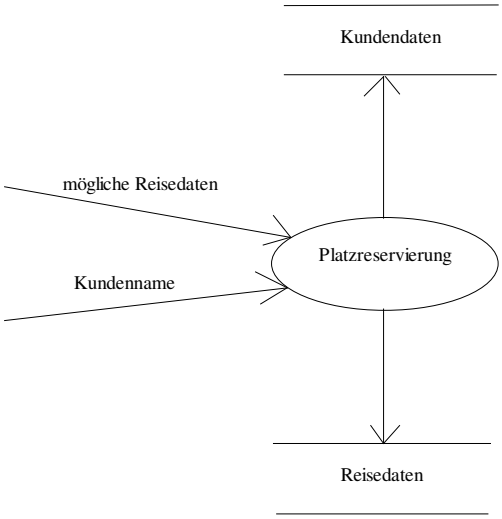


Abbildung 66: DFD Platzreservierung

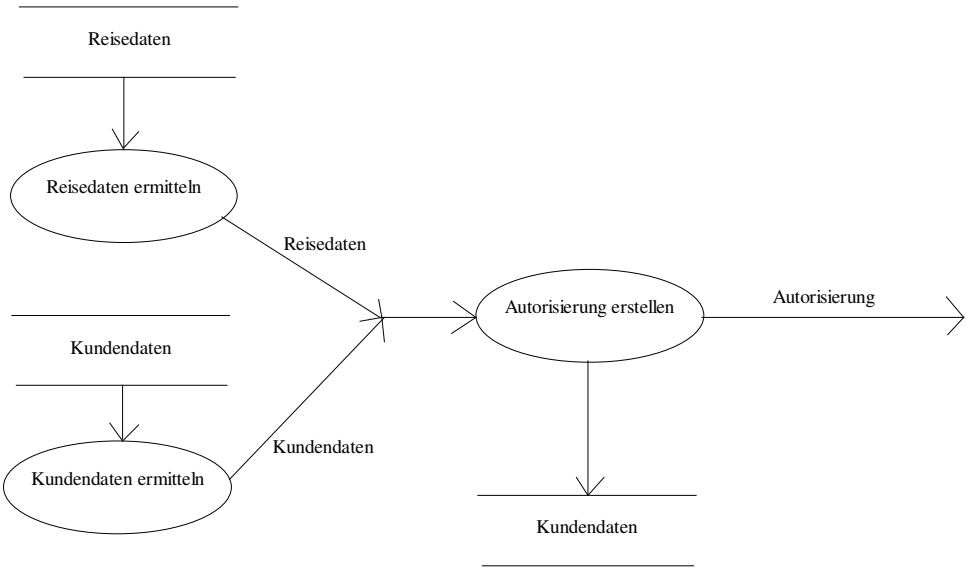


Abbildung 67: DFD Autorisierung erstellen

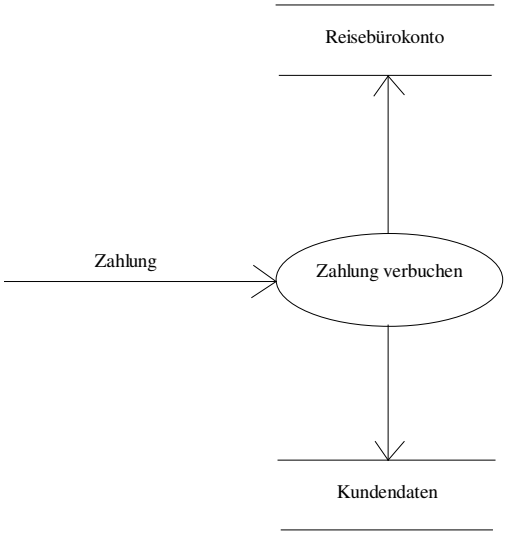


Abbildung 68: DFD Zahlung verbuchen

Die Beschreibung der Funktionen kann aus mathematischen Funktionen, natürlicher Sprache, Pseudocode oder anderen Beschreibungsformen bestehen. Hier wurde die natürlichsprachliche Alternative gewählt. Es wird beschrieben, was die Funktion berechnet. Eine Implementation wird hier nicht vorgenommen.

- Broschüre aushändigen
  - Eine Broschüre wird dem Kunden ausgehändigt.
- Reisedaten überprüfen
  - Aus den verfügbaren Reisen werden die ausgewählt, die den gewünschten Reisedaten entsprechen oder die Daten möglichst gut erfüllen. Das Ergebnis der Überprüfung sind die möglichen Reisedaten, d.h. die Busreisen, bei denen ein freier Reiseplatz verfügbar ist.
- Kundendaten erfassen/aktualisieren
  - Bucht ein Kunde zum ersten Mal eine Reise, so werden seine persönlichen Daten erfaßt. Sind die Kundendaten dem Unternehmen bereits bekannt, so werden diese gegebenenfalls aktualisiert, falls sich an diesen etwas geändert hat.
- Platzreservierung
  - Bei den Reisedaten wird bei der ausgewählten Reise die Anzahl der gebuchten Plätze um ‘1’ erhöht. Der Name des Kunden wird hier nicht erfaßt.
    Bei den Kundendaten werden die Daten der gebuchten Reise in die Liste der gebuchten Reisen bei dem entsprechenden Kunden eingetragen.
- Zahlung verbuchen
  - In den Kundendaten wird die Bezahlung einer gebuchten Reise vermerkt. Dazu wird auch das Datum der Bezahlung eingetragen.
- Reisedaten ermitteln
  - Ermittelt werden die Daten zu einer bestimmten Busreise. Diese stehen im Speicher der Reisedaten.
- Kundendaten holen
  - Ermittelt werden die Daten eines bestimmten Kunden aus dem Kundendatenspeicher.
- Autorisierung erstellen

  - Eine Autorisierung wird aus den ermittelten Reise- und Kundendaten erstellt und vermerkt. Der Kunde erhält eine Kopie der Autorisierung.
- Autorisierung entwerten
  - Die Autorisierung wird vom Kunden vorgelegt und entwertet. Dabei wird die Entwertung im System vermerkt.
- Bordkarte ausgeben
  - Die Bordkarte wird nach der Autorisierungsentwertung ausgegeben.
- Gepäckanhänger ausgeben
  - Die Gepäckanhänger werden nach der Autorisierungsentwertung ausgegeben.

3.3 Spezifikation von Minitours mit UML

Bei dem Einsatz der Unified Modeling Language (UML) zur Modellierung des Reiseunternehmens werden verschiedene Diagramme entwickelt, die in ihrer Gesamtheit das System beschreiben. Da eine hohe Wechselwirkung zwischen den Diagrammen beim Entwurf stattfindet, ist die Reihenfolge, in der sie aufgeführt werden, nicht entscheidend. Wesentlich ist, daß Veränderungen in einem Diagramm andere Diagramme in der Regel beeinflussen. Durchgeführt wird also, wie bei dem Einsatz von OMT, ein iteratives Vorgehen. Auch in diesem Abschnitt wird wegen der Übersichtlichkeit des Fallbeispiels keine spezielle Methode angewendet.

Beeinflußt wurde die Entwicklung durch den Einsatz des Modellierungswerkzeugs ‘Rational Rose’. Vorgestellt wird das Use-Case-Diagramm, das aus der textuellen Systembeschreibung hervorgeht. Das Klassendiagramm beschreibt die benötigten Klassen und Assoziationen zwischen diesen. In den Interaktionsdiagrammen wird die Kommunikation zwischen den Objekten dargestellt. Zustandsdiagramme visualisieren die dynamischen Aspekte der Objekte. Das verwendete Werkzeug unterstützt die Erhaltung der Konsistenz des Modells bei Änderungen. Die Erstellung von Aktivitätsdiagrammen wird z.Zt. noch nicht von Rational Rose unterstützt und wurde deshalb nicht durchgeführt. Die Diagramme sind alle konsistent in englischer Sprache gehalten, um diese für Diskussionen verwenden zu können, die über diese Arbeit hinaus eventuell im englischsprachigen Raum geführt werden.

3.3.1 Use-Case-Diagramm

Das Use-Case-Diagramm in der Abbildung 69 zeigt die Verwendung des Systems und die Akteure zu diesen Anwendungsfällen. Zum Beispiel ist der ‘Administrator’ bei dem Use-Case ‘System Startup’ der Akteur. Analog werden die anderen Akteure und Use-Cases betrachtet. Der Manager registriert sich via Login Use-Case im System, kann Reiseschemata und Busreisen erstellen, sowie Busse eintragen. Ein Kunde (Client) kann sich ebenfalls bei dem System anmelden, sich über Reiseschemata und Busreisen informieren und Reisen buchen. Ein Manager kann die Rolle eines Kunden annehmen. In dem Moment stehen ihm nur die Kundenrechte zur Verfügung. Ebenso kann ein Kunde die Rolle eines Managers übernehmen, wenn er dazu autorisiert wird. Ein Mitarbeiter (CheckInStaff) des Reiseunternehmens ist gemeinsam mit dem Kunden bei der Abfertigung (Check-In) aktiv. Auch der Mitarbeiter meldet sich beim System an.

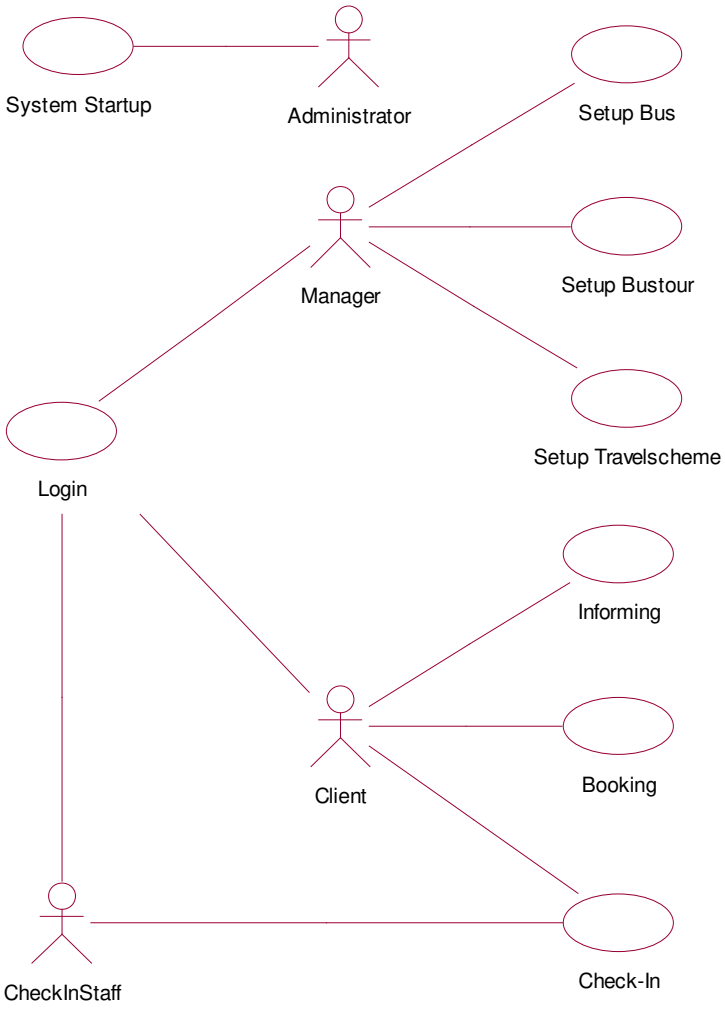


Abbildung 69: Use-Case-Diagramm

3.3.2 Klassendiagramm

In diesem Abschnitt erfolgt die Betrachtung des Klassendiagramms für Minitours, das die Klassen zeigt, die zur Realisierung der Use-Cases benötigt werden. Im folgenden Text wird das Klassendiagramm aus der Abbildung 70 dokumentiert. Bei den Akteuren beginnend werden die Klassen, die Assoziationen zwischen ihnen und die Multiplizitäten erläutert. Das Diagramm ist aus technischen Gründen an dieser Stelle etwas klein geraten und der Leser möge bitte der besseren Lesbarkeit wegen dieses Diagramm im ersten Teil des Anhangs nachschauen. Bei der textuellen Beschreibung der Multiplizitäten steht ‘mehrere’ für ‘ein’ oder ‘viele’.

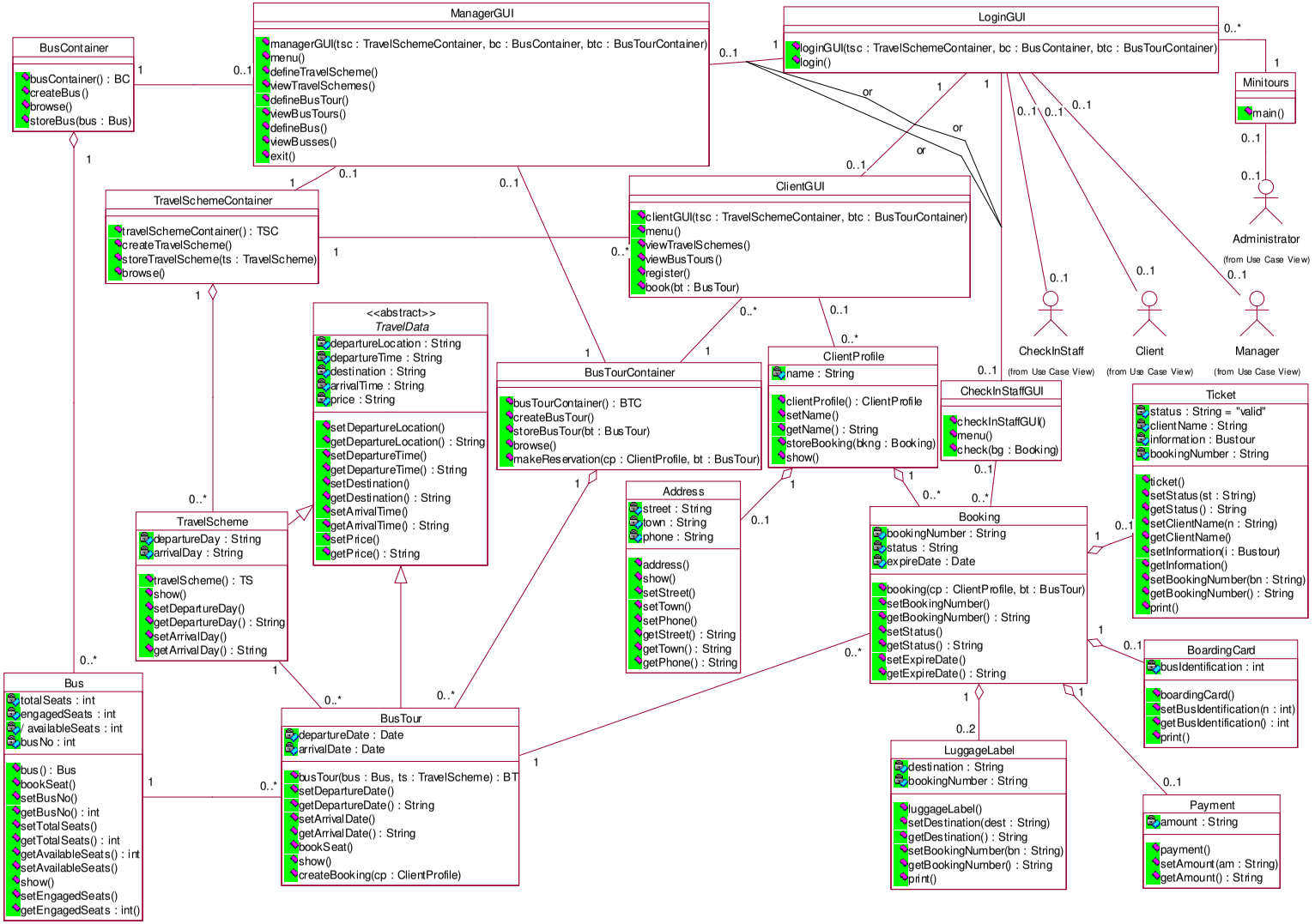


Abbildung 70: Klassendiagramm

Die Klasse ‘Minitours’ enthält die statische Methode ‘main()’. Diese wird vom Administrator aufgerufen, um das System hochzufahren. Zu einem Objekt der Klasse ‘Minitours’ kann es kein oder mehrere Objekte vom Typ ‘LoginGUI’ geben.

Zu einem LoginGUI muß es genau ein Objekt der Klasse Minitours bzw. einen Aufruf einer main() Methode geben. Zu einem LoginGUI kann es keinen oder einen Akteur geben. Je nach der Art des Logins gibt es zu einem LoginGUI ein ManagerGUI oder ein ClientGUI oder ein CheckInStaffGUI. Zu jedem dieser letztgenannten GUIs muß es genau ein LoginGUI geben.

Zu einem ManagerGUI muß es genau einen BusContainer, einen BusTourContainer und einen TravelSchemeContainer geben. Zu diesen Containern muß kein, kann aber höchstens ein ManagerGUI existieren.

Der BusContainer aggregiert keine oder mehrere Busse und jeder Bus wird von genau einem BusContainer aggregiert. Der BusTourContainer aggregiert keine oder mehrere BusTours und eine BusTour wird von genau einem BusTourContainer aggregiert. Ein Bus kann für keine, eine oder viele BusTours eingesetzt werden. Zu einer Bustour gibt es immer genau einen für diese BusTour eingesetzten Bus. Ein TravelSchemeContainer aggregiert keine oder mehrere TravelSchemes. Zu einem TravelScheme kann es keine oder mehrere BusTours geben, während es zu einer BusTour genau ein TravelScheme geben muß. TravelScheme erbt, wie auch BusTour, von der abstrakten Klasse TravelData.

Zu einem TravelSchemeContainer und einem BustourContainer kann es alternativ zu dem ManagerGUI kein oder ein ClientGUI geben. Zu einem ClientGUI kann es keine, ein oder viele ClientProfiles geben. Zu jedem ClientProfile kann es temporär das zu dem angemeldeten Kunden gehörende ClientGUI geben. Ein ClientProfile aggregiert keine oder mehrere Bookings und keine oder eine Adresse. Ein Booking muß von genau einem ClientProfile aggregiert werden. Ein Booking bezieht sich auf genau eine BusTour, während es zu einer BusTour viele Bookings geben kann. Zu einem Booking kann es kein oder ein CheckInStaffGUI und zu diesem kann es keine oder mehrere Bookings geben.

Ein Booking aggregiert kein oder ein Ticket, BoardingCard und Payment und höchstens zwei LuggageLabels. Jedes dieser Aggregate muß zu genau einem Booking gehören.

3.3.3 Sequenzdiagramme

Die Erzeugung von Interaktionsdiagrammen wird von dem Werkzeug ‘Rational Rose’ sehr gut unterstützt. In den folgenden Abbildungen sind die Methodenaufrufe dargestellt. Es sei daran erinnert, daß die Zeit in diesen Diagrammen von oben nach unten verläuft. Methoden, die wie die Klassen benannt sind, bei dessen Objekten sie aufgerufen werden, erzeugen Objekte dieser Klassen. Zum Beispiel erzeugt der Aufruf der Methode ‘TravelSchemeContainer()’ ein Objekt der Klasse ‘TravelSchemeContainer’. Bei dem Aufruf eines solchen Konstruktors können Parameter übergeben werden. Das Zeichen ‘*’ vor einem Methodenaufruf steht für Iteration.

In der Abbildung 71 sind die bei dem Systemstart erfolgenden Methodenaufrufe dargestellt. Bei dem Objekt Minitours wird die Methode ‘main()’ aufgerufen. Dieser Aufruf veranlaßt die Aufrufe der Konstruktormethoden für einen TravelSchemeContainer, BusContainer und einen BusTourContainer. Die Reihenfolge der Erzeugung dieser Containerobjekte ist beliebig. UML sieht konkurrente Methodenaufrufe vor, dies wird grafisch von ‘Rational Rose’ allerdings nicht unterstützt, deshalb sind die Methodenaufrufe in diesem Diagramm sequentiell. Nach der Erzeugung der Container wird ein Objekt vom Typ ‘LoginGUI’ erzeugt. Bei diesem ruft der Administrator die Methode ‘login()’ auf. Das System befindet sich dann in dem Initialzustand und die Methode ‘login()’ des LoginGUIs wartet auf eine Benutzereingabe.

In der Abbildung 72 ist das Anmelden eines Managers dargestellt. Dieser interagiert mit dem Login-GUI, indem er die benötigte Eingabe liefert. Dann wird der Konstruktor der Klasse ‘ManagerGUI’ aufgerufen und somit ein Objekt dieses Typs erzeugt. Dieses Objekt ruft bei sich selbst die Methode ‘menu()’ auf und wartet auf eine Benutzereingabe.

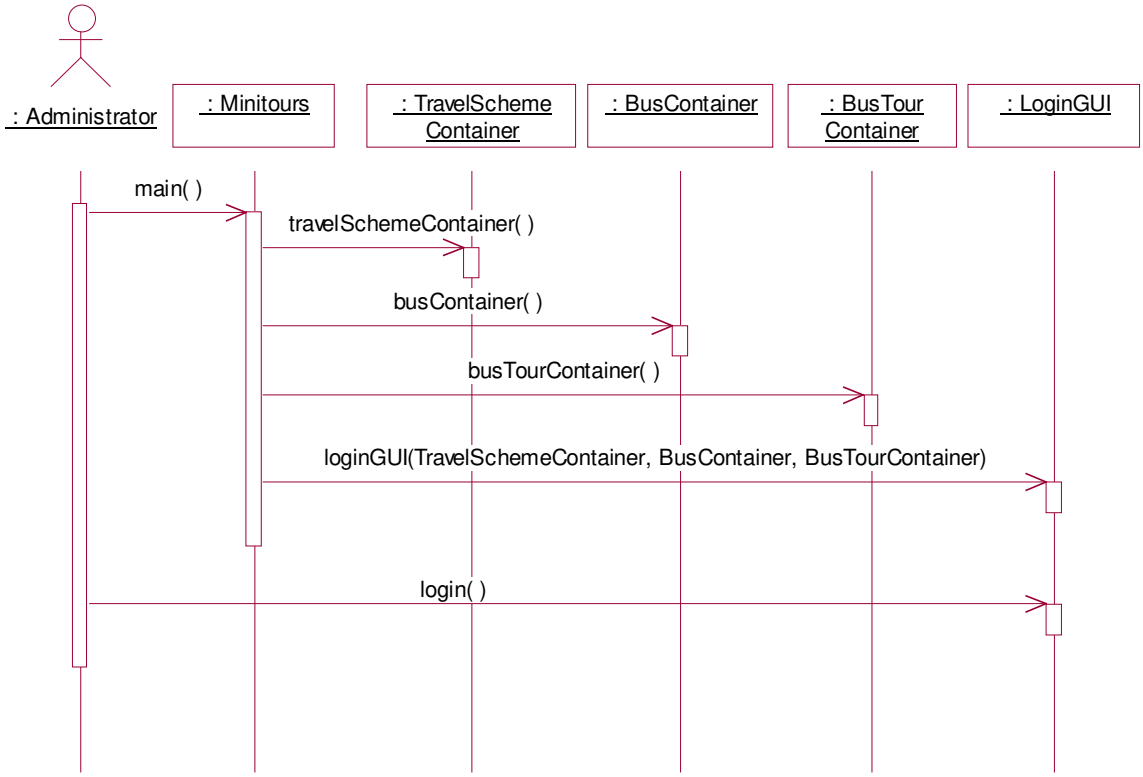


Abbildung 71: Start des Systems

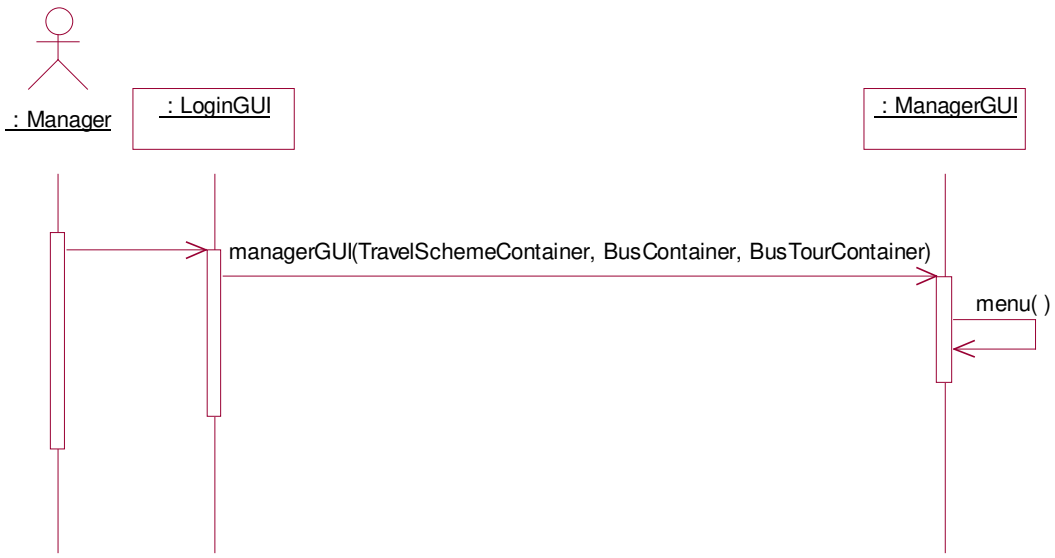


Abbildung 72: Manager-Login

Die Abbildung 73 zeigt den Ablauf der Methodenaufrufe, wenn die als Manager angemeldete Person ein neues Reiseschema definiert. Entsprechend der Benutzereingabe bei der auf die Eingabe wartende Methode ‘menu()’ wird die Methode ‘createTravelScheme()’ des ManagerGUIs aufgerufen. Diese wiederum ruft den Konstruktor der Klasse ‘TravelScheme’ auf. Dieser ruft die set-Methoden des neuen TravelScheme-Objekts auf, um die Eingaben für die benötigten Werte zu bekommen. Die Eingabe dieser Werte kann in beliebiger Reihenfolge ablaufen und ist hier nur wegen des verwendeten Werkzeugs sequentiell dargestellt. Nach der Erzeugung eines TravelScheme-Objekts wird dessen Referenz an das TravelSchemeContainer-Objekt zurückgegeben, um diese Referenz bei diesem durch die Methode ‘storeTravelScheme(ts:TravelScheme) ’ zu speichern. Analoge Abläufe gibt es bei der Definition von Bussen und BusTours.

Abbildung 73: Definieren eines Reiseschemas

Das Durchsehen der TravelSchemes wird durch die Abbildung 74 spezifiziert. Durch den Aufruf der Methode ‘viewTravelSchemes()’ beim ManagerGUI durch die Benutzereingabe wird bei dem TravelSchemeContainer die Methode ‘browse()’ aufgerufen. Diese ruft iterativ alle show()-Methoden

der einzelnen ihm bekannten TravelSchemes auf. Eine show()-Methode ist dafür verantwortlich, die Werte eines TravelSchemes zu präsentieren. Analog verläuft dies für das Durchsehen der Busse und der BusTours.

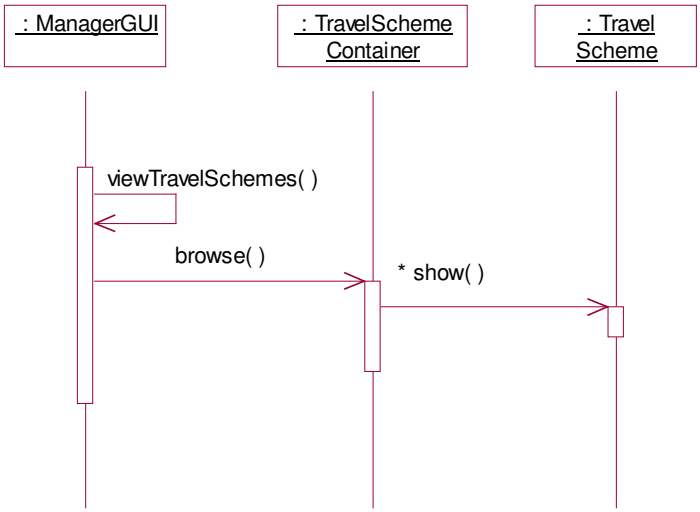


Abbildung 74: Durchsehen der Reiseschemata

Die Methodenaufrufe bei einer Kundenanmeldung sind analog zu der Anmeldung eines Managers. Nachdem der Kunde die Eingabeaufforderung erfüllt hat, wird ein Kundeninterface-Objekt vom Typ ‘ClientGUI’ erzeugt, welches nach der Erzeugung bei sich die Methode menu() aufruft. Diese wartet auf eine Benutzereingabe. Für den Fall, daß sich der Kunde die TravelSchemes ansehen möchte, erfolgen die Methodenaufrufe wie bereits in der Abbildung 74 dargestellt. Möchte der Kunde eine Reise buchen, so wählt er den entsprechenden Menüpunkt und das Szenario aus der Abbildung 76 beginnt. Sofern der Kunde im System noch nicht bekannt ist, muß er sich registrieren. Dies erfolgt wie in der Abbildung 77 gezeigt. Das Anmelden eines Mitarbeiters erfolgt analog zu den Anmeldevorgängen der Manager und Kunden. Bei der Abfertigung erfolgen die Methodenaufrufe aus der Abbildung 78.

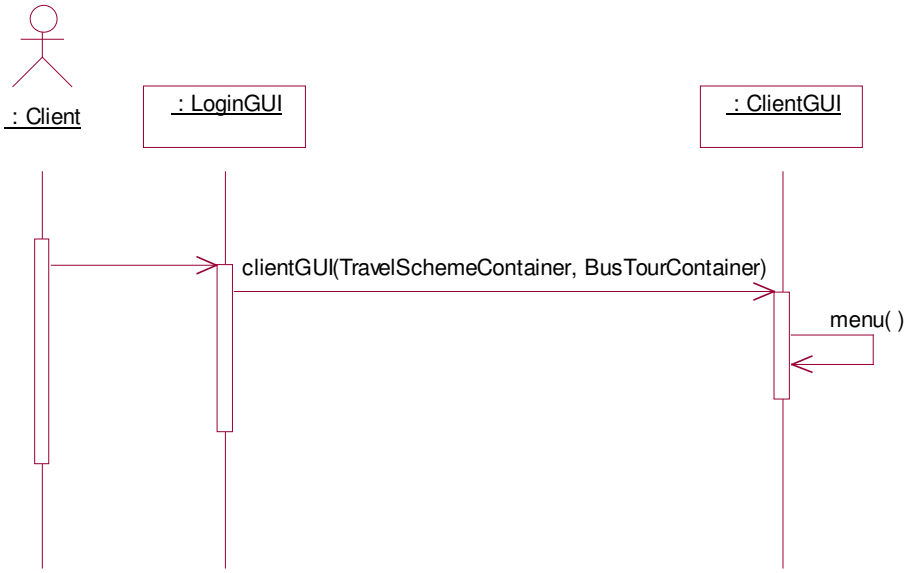


Abbildung 75: Kunden-Login

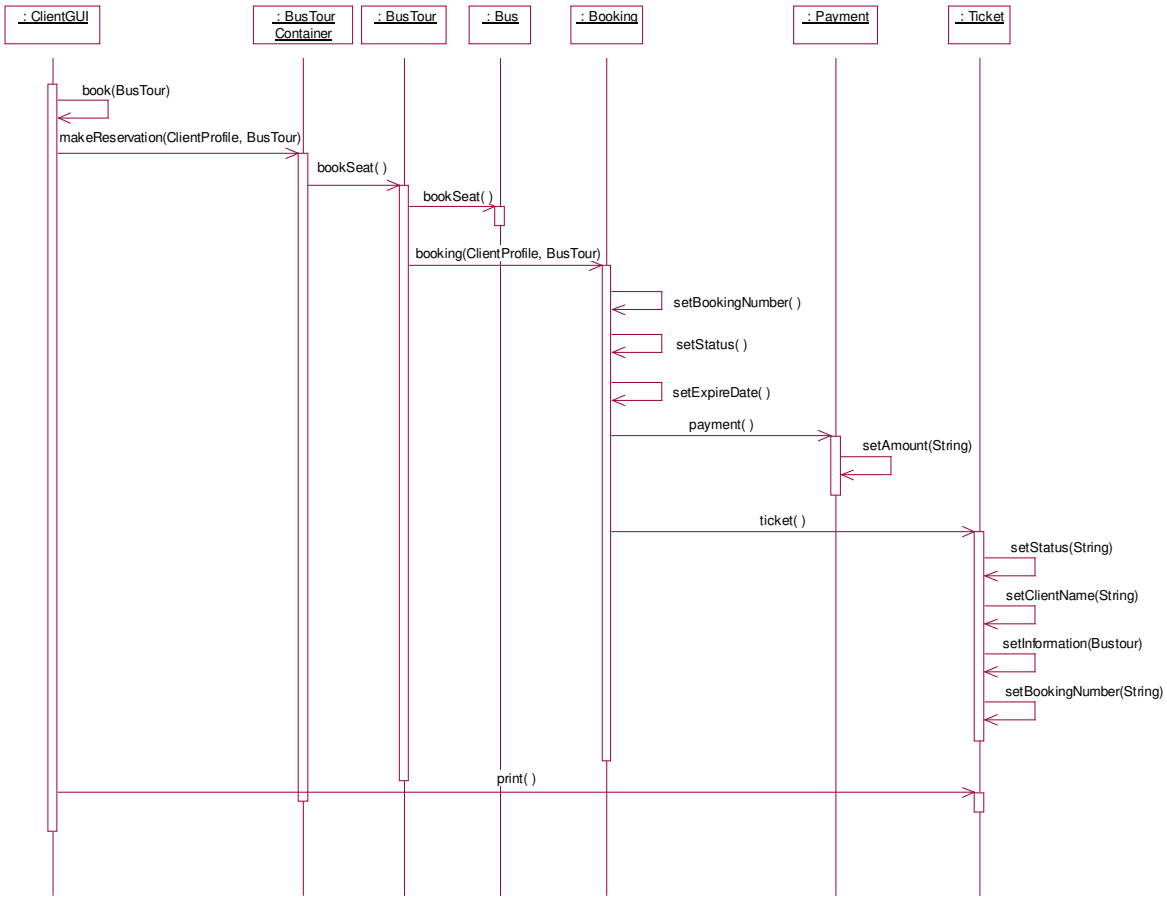


Abbildung 76: Buchung

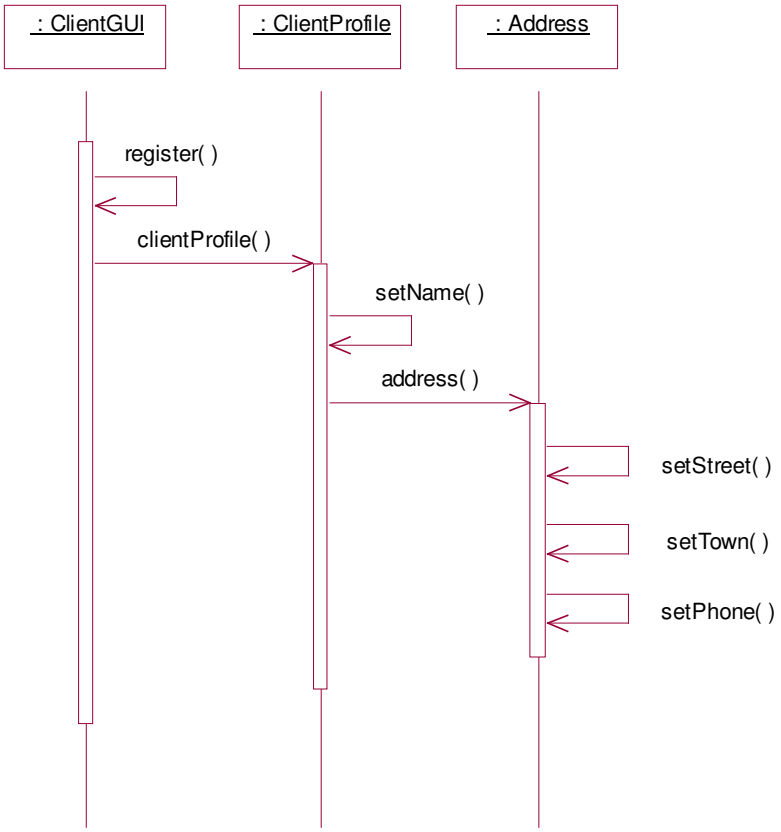


Abbildung 77: Kundenregistrierung

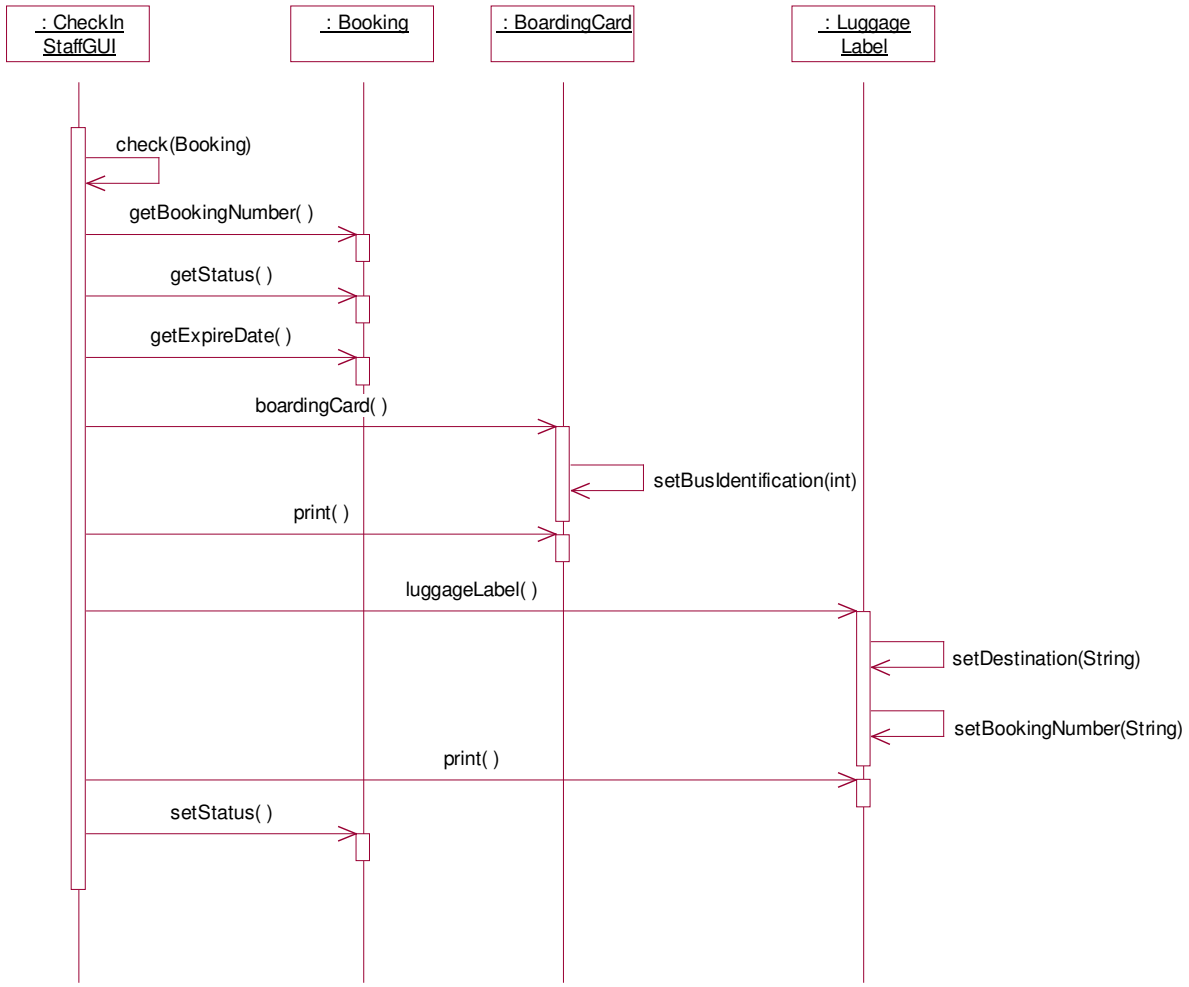


Abbildung 78: Abfertigung

3.3.4 Zustandsdiagramme

In diesem Abschnitt werden einige Zustandsdiagramme vorgestellt. Sie zeigen die möglichen Zustände der Objekte von einigen Minitours-Klassen. Das Zustandsdiagramm in der Abbildung 79 zeigt die Zustände eines LoginGUIs. Durch den Aufruf der Methode ‘login()’ ist ein Zustand erreicht, in dem auf eine Benutzereingabe gewartet wird. Sofern eine gültige Eingabe erfolgt, wechselt der Zustand. Ein Benutzer ist nun im System angemeldet. Sobald sich dieser abmeldet, ändert sich der Zustand wieder. Es besteht nun die Möglichkeit, durch eine Benutzereingabe das System herunterzufahren oder wieder in den Zustand überzugehen, in dem eine Benutzeranmeldung erwartet wird.

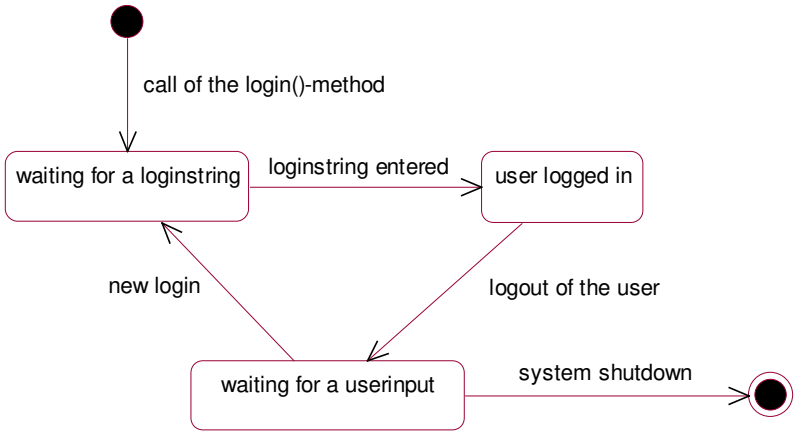


Abbildung 79: Zustandsdiagramm für ein LoginGUI

Ein ManagerGUI geht, ähnlich dem LoginGUI, nach der Erzeugung in einen Zustand über, in dem eine Eingabe vom Benutzer erwartet wird. Wählt dieser eine Funktion aus, so verändert sich der Zustand des ManagerGUIs. Wenn der Benutzer das Menü durch die Auswahl der exit-Funktion beendet, dann wird das ManagerGUI eliminiert.

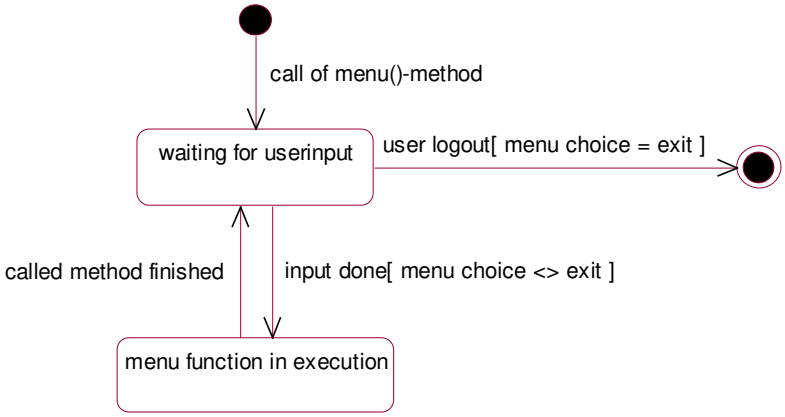


Abbildung 80: Zustandsdiagramm für ein ManagerGUI

Die Abbildung 81 zeigt das Zustandsdiagramm eines BusContainers. Nach der Erzeugung ist dieser zunächst leer. Sobald ein Bus hinzugefügt wird ist der Container nicht mehr leer. Der Container bleibt in diesem Zustand, auch solange Busse hinzugefügt werden, bis das System heruntergefahren wird. Das Herunterfahren veranlaßt die Eleminierung des Containers. Das Löschen einzelner Busse ist nicht vorgesehen, daher kann der Zustand ‘empty’ nicht wieder erreicht werden.

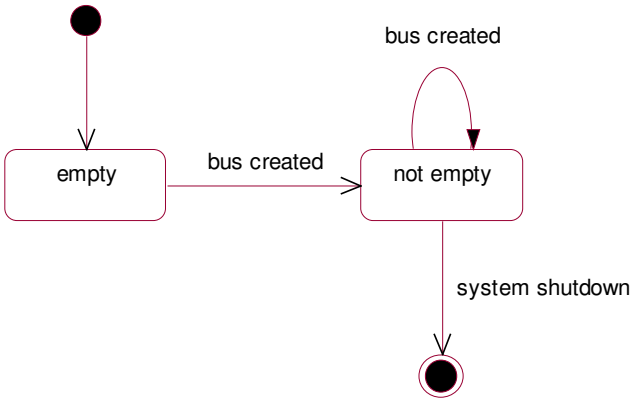


Abbildung 81: Zustandsdiagramm für einen BusContainer

Ein Bus befindet sich nach der Erzeugung in dem Zustand, in dem alle Sitzplätze verfügbar sind (Abbildung 82). Wird der erste Sitzplatz gebucht, so erfolgt ein Zustandswechsel. Solange die Anzahl der freien Plätze größer als Null ist verändert sich der Zustand nicht. Erst wenn der letzte Platz gebucht wird, wird der Zustand ‘no seats available‘, d.h. keine freien Plätze mehr, erreicht. Aus jedem dieser Zustände kann der Endzustand erreicht werden, wenn das die Busse aggregierende Objekt eliminiert wird.

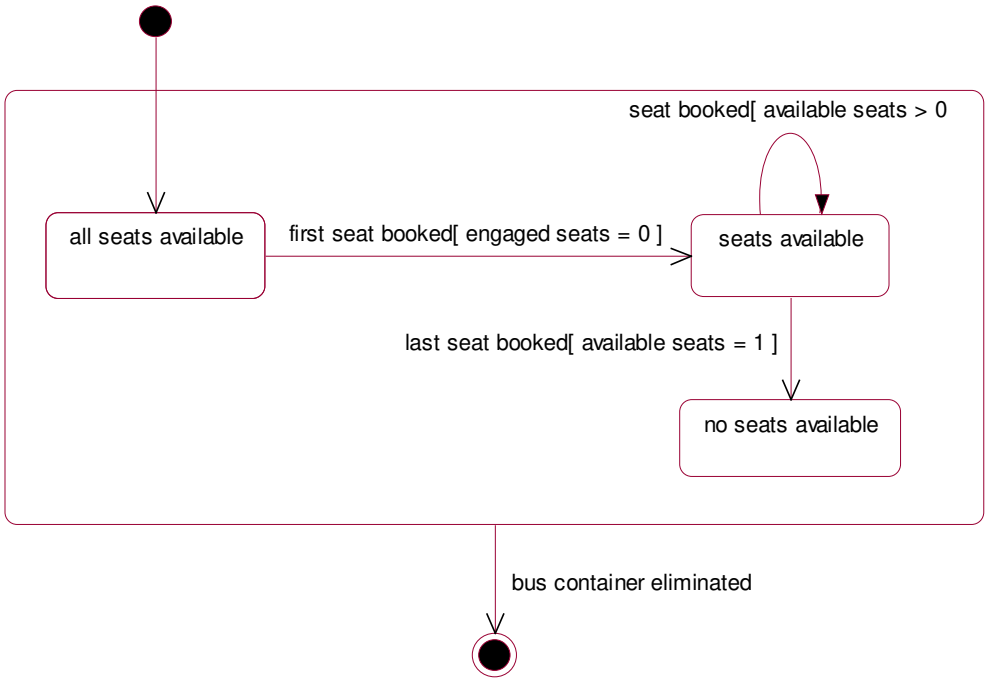


Abbildung 82: Zustandsdiagramm für einen Bus

Eine Buchung kann sich in einem der in Abbildung 83 dargestellten Zustände befinden. Zuerst hat eine Buchung den Status ‘valid’, d.h. sie ist gültig und kann potentiell genutzt werden, um die Reise anzutreten. Geschieht dies nicht vor Ablauf der Gültigkeitsfrist, so verfällt die Gültigkeit und die Buchung befindet sich in dem entsprechenden Zustand.

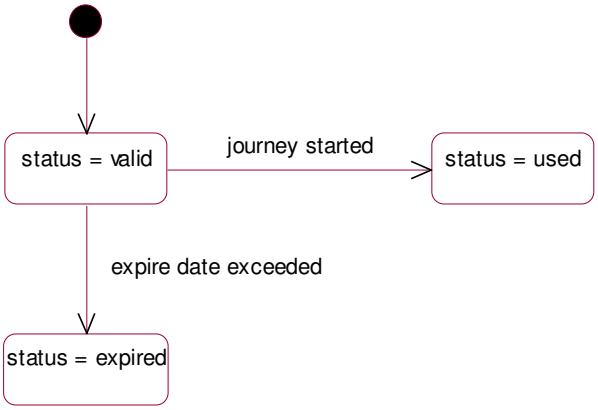


Abbildung 83: Zustandsdiagramm für eine Buchung

Weitere Zustandsdiagramme werden an dieser Stelle nicht explizit aufgeführt, da sie meiner Ansicht nach zu trivial sind. Objekte der Klassen TravelScheme, BusTour, ClientProfile, Address, Ticket, BoardingCard, Payment und LuggageLabel werden zu einem Zeitpunkt erzeugt, existieren dann eine Zeitlang und werden beim Herunterfahren des Systems vernichtet. Zur persistenten Datenhaltung könnten die Daten bei dem Herunterfahren des Systems auf einen sekundären Speicher ausgelagert und beim Hochfahren des Systems geladen werden. Dies wird im allgemeinen so gemacht, ist hier aber nicht realisiert worden, da der Fokus bei diesem Anwendungsbeispiel auf der Modellierung der dynamischen Abläufe des Anwendungsgebietes liegt.

3.4 Ergebnisse der objektorientierten Analyse

Der wesentliche Unterschied bei der Entwicklung der beiden Modelle in diesem Kapitel liegt in der Vorgehensweise. Während bei dem Einsatz von OMT die Datenmodellierung als Ausgangspunkt für die Analyse dient, werden bei dem Einsatz von UML zuerst Use-Cases betrachtet. Eine Bewertung, welche der Vorgehensweisen besser ist, kann im Rahmen dieses kleinen Beispiels nicht vorgenommen werden. Der Einsatz von ‘Rational Rose’ als Modellierungswerkzeug hat den Entwicklungsprozeß sehr gut unterstützt. Die bei den Iterationen vorgenommenen Änderungen in den Klassen- und Interaktionsdiagrammen werden konsistent gehalten. Beiden Modellen ist gemeinsam, daß die Klassendiagramme direkt zur Codegenerierung verwendet werden können, während die anderen Diagramme nur dokumentarischen Charakter haben.

Bei den Analysemodellen handelt es sich nicht um ausführbare Spezifikationen und somit kann das modellierte Systemverhalten nicht prototypisch getestet werden.

Die Modelle des Reiseunternehmens Minitours liefern die Basis für die Entwicklung der ausführbaren Spezifikation mit Petrinetzen. Die Klassendiagramme zeigen neben den benötigten Klassen deren Beziehungen zueinander. Die Interaktionsdiagramme werden benötigt, um die dargestellten Szenarien Schrittweise zu implementieren. Die Zustandsdiagramme helfen bei der Implementation der Methoden.

4 OBJEKTORIENTIERTE KONZEPTE FÜR PETRINETZE

In diesem Kapitel werden unterschiedliche Ansätze zur Umsetzung objektorientierter Konzepte für Petrinetze vorgestellt. Jeder Ansatz realisiert die aus der objektorientierten Programmierung bekannten Konzepte der Klassen, der Klassenvariablen und der statischen Methoden und der Objekte mit ihren Attributen und Methoden. Weitere Konzepte wie Vererbung, Interfaces, Assoziationen und Polymorphismus werden nicht bei allen Petrinetz-Ansätzen umgesetzt bzw. erfahren eine unterschiedliche Gewichtung. Zur Entwicklung der Spezifikation werden iterative Vorgehensweisen vorgeschlagen. Auf das zur Modellierung und Simulation der Petrinetze verwendete Werkzeug wird am Ende dieses Kapitels eingegangen.

4.1 Historie

Ansätze, Petrinetze mit objektorientierten Konzepten zu verbinden, findet man u.a. in [Buchs und Guelfi 91], [Becker und Moldt 93a], [Becker und Moldt 93b], [Bruno 94], [Lakos 94], [Lakos 95], [Maier 96], [Moldt 96], [Maier 97] und [KMW 98]. Das gemeinsame Ziel aller Ansätze ist die Bereitstellung einer Notation zur Modellierung von ausführbaren Systemspezifikationen.

Buchs und Guelfi schlagen in [Buchs und Guelfi 91] einen Formalismus namens CO-OPN (Concurrent Object-Oriented Petri Nets) vor, bei dem eine algebraische Spezifikation eingesetzt wird, um Objekte zu beschreiben. Klassen werden in CO-OPN durch diese algebraische Spezifikationssprache und gefärbte Petrinetze beschrieben. Die Kommunikation zwischen Klassen erfolgt über Transitionsverschmelzungen. Objekte werden jedoch statisch und nicht dynamisch zur Laufzeit erzeugt.

In [Becker und Moldt 93a] und [Becker und Moldt 93b] werden Einschränkungen an den gefärbten Petrinetzen nach [Jensen 92] vorgenommen. Diese Einschränkungen betreffen die Struktur (statische Objekt- und Klassenstruktur), inklusive der Farbmengen und Beschriftungen der Netze und die Marken (Nachrichten), die zwischen den Netzen ausgetauscht werden. Die resultierenden Netze werden als Object Oriented Coloured Petri Nets (OOCPN) bezeichnet.

Ein Ansatz, der Protob-Netze genannt wird, ist in [Bruno 94] beschrieben. Protob-Netze können nach objektorientierten Gesichtspunkten strukturiert werden. Ein Netz kommuniziert über Eingangs- und Ausgangsstellen mit seiner Umgebung. Objekte können zu neuen Objekten zusammengesetzt und mit weiteren Netzelementen ergänzt werden. Dadurch wird die Bildung komplexer Objekte möglich.

Einen neuen Formalismus für Object Petri Nets (OPNs) hat Lakos in [Lakos 94] und [Lakos 95] vorgestellt. In diesem Formalismus kapseln Objekte ihre Attribute und Operationen und verfügen über klare Schnittstellen. Zwischen Klassen können Vererbungsbeziehungen definiert werden. Es können so Systeme von selbständig agierenden Objekten modelliert werden, die über Nachrichten kommunizieren. Objekte können selbst Objekte enthalten, die ebenfalls aktiv sind. Netze sind die allgemeine Kontrollstruktur innerhalb derer sich die Objekte, in Form von Marken, bewegen.

In [Moldt 96] werden die aus der objektorientierten Programmierung bekannten Konzepte für Petrinetze definiert. Vorgestellt werden u.a. Klassen- und Objektnetze, Attribute, Methoden, Vererbung und Assoziationen. Dieser Ansatz wird im Laufe dieser Arbeit ausführlich erläutert.

Maier schlägt in [Maier 97] einen neuen Formalismus vor, der die von ihm definierten Object Coloured Petri Nets (OCPN) vorstellt. Bei diesen Netzen ist das Klassennetz das Erzeugungsmuster für konkrete Objektnetze. Bei der Instanziierung eines Objekts werden Stellen, Transitionen, Kantenin-

stanzen und die Initialmarkierung für das Objektnetz entsprechend dem definierenden Klassennetz angelegt. Dieses Konzept ist analog zur objektorientierten Programmierung, denn dort bilden die Klassen die Muster für die zur Laufzeit instanziierten Objekte. In den OCP-Netzen wird zur Ausführungszeit für jedes Objekt ein eigenes Objektnetz erzeugt, indem Instanzen für Stellen, Transitionen und Kanten angelegt werden.

Die in dieser Arbeit vorgestellten objektorientierten Petrinetze basieren auf den Vorschlägen von Maier und Moldt. Die objektorientierten Petrinetze setzen die Konzepte aus der objektorientierten Programmierung um, indem Objekte und Klassen als Netze dargestellt werden. Nachrichten entsprechen Marken. Es werden die verwendeten Konzepte erläutert und unterschiedliche Konstrukte zu deren Darstellung gegenübergestellt.

Die Verwendung des Werkzeuges Design/CPN bestimmt die technischen Randbedingungen für die Konstrukte. Unterstützt wird die hierarchische Modellierung. Diese fordert allerdings bei der mehrfachen Verwendung von Unterseiten eine etwas ‘eigenwillige’ Umsetzung. Im Laufe dieser Arbeit wird darauf eingegangen. Ein weiterer Einfluß der gewählten Software ist die Verwendung der funktionalen Programmiersprache ML. Diese wird zur Definition für den in den Netzen benötigten Code eingesetzt. Dieser wird z.B. für Kantenbeschriftungen und Code-Regionen der Transitionen benötigt. Informationen zu Petrinetzen und der Verwendung der Software Design/CPN findet man im Internet unter ‘http://www.daimi.au.dk/designCPN’.

4.2 Objekt- und Klassennetze

Die Konzepte der objektorientierten Programmierung werden u.a. in [Meyer 90] und [Louden 94] beschrieben. Objekte kapseln ihre Daten und Operationen, die auf diesen Daten ausgeführt werden. Die Operationen (Methoden) sind nur über Nachrichten aufrufbar. Objekte besitzen einen veränderbaren lokalen Zustand, der nur durch den Zugriff auf die objekteigenen Methoden verändert werden kann. Ein Objekt besitzt eine eindeutige Identität. Diese ist wichtig, um ein Objekt als Empfänger oder Absender einer Nachricht identifizieren zu können. In diesem Kapitel wird die Darstellung von Klassen und Objekten als Petrinetze vorgestellt.

4.2.1 Objekte als Petrinetze

In [Moldt 96] werden Objekte als Objektnetze wie in Abbildung 84 dargestellt. Sie werden Object-Oriented Coloured Petri Nets (OOCPN) genannt. Die Attribute, aus der programmiersprachlichen Sicht als Instanzvariablen bezeichnet, werden als Stellen `inst_var` und die Methoden als Transitionen `method 1..n` realisiert. Nachrichten an ein Objekt werden aus der Stelle `in_pool` entnommen. Um sicherzugehen, daß die Nachrichten für dieses Objekt sind, wird durch die Transition `input` der Empfänger überprüft. Entspricht die Empfänger-ID in der Nachricht der Objekt-ID in der Stelle `own_id`, so ist eine Nachricht für das Objekt erkannt worden und wird in die Stelle `rec-msg` weitergereicht. Eine Methode erkennt, durch die Verwendung einer Schutzbedingung (Guard), die für sie bestimmte Nachricht und die entsprechende Transition kann schalten. Ein Guard ist also eine Bedingung, die für eine Transition zum Schalten erfüllt sein muß. In diesem Fall muß der Methodenname in der Nachricht mit dem Methodennamen im Guard der entsprechenden Transition übereinstimmen. Nur die Methoden-Transitionen eines Objekts und Methoden der Klasse, wie z.B. die im folgenden Unterkapitel eingeführte Initialisierungsmethode `new` des Klassennetzes, können auf die Attribute (Instanzvariablen) zugreifen. Die von der Methoden-Transition erzeugten Nachrichten gelangen in die Stelle `send_msg`. Die Transition `output` schaltet und legt die Nachrichten aus der Stelle `send_msg` in die Stelle `out_pool`. Die Kommunikation zwischen Objekten wird

später ausführlich beschrieben. Aus der Abbildung 84 ist zunächst zu entnehmen, daß ein Objekt über die Stellen `in_pool` und `out_pool` mit anderen Objekten kommuniziert.

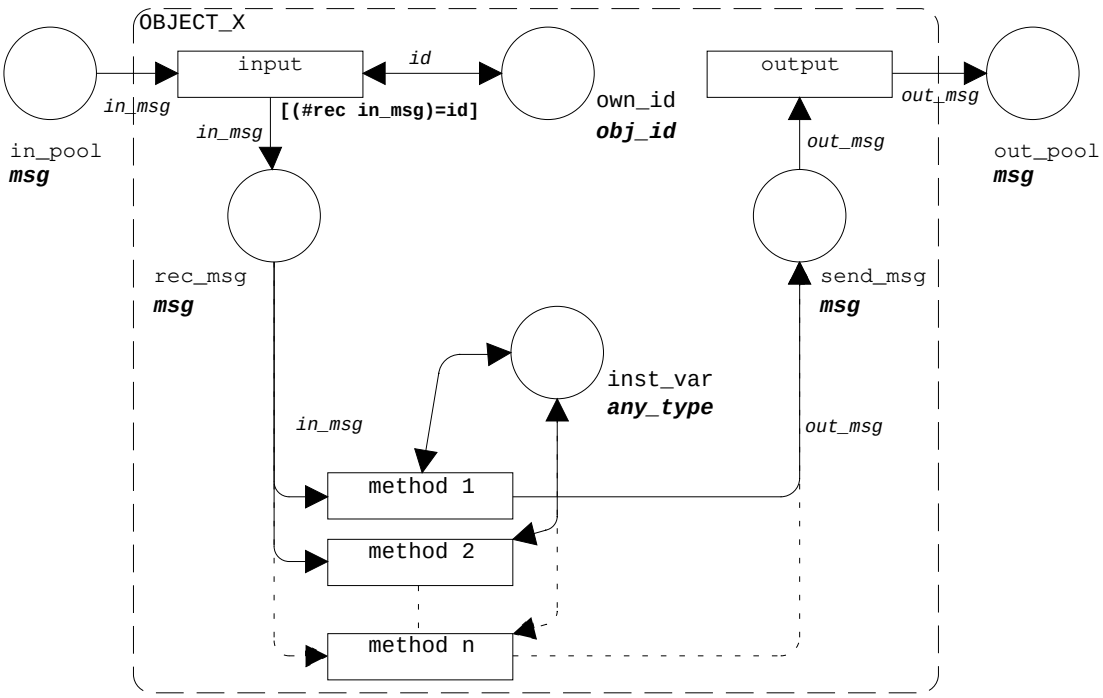


Abbildung 84: Objektnetz aus [Moldt 96]

4.2.2 Klassen als Petrinetze

Klassennetze definieren die Struktur und das Verhalten für eine Menge von Objekten, die Exemplare dieser Klasse sind. Die Abbildung 85 zeigt ein Klassennetz aus [Moldt 96]. Klassennetze können als Faltung von Objektnetzen betrachtet werden, die für die Erzeugung, Verwaltung und Vernichtung von Objekten zuständig sind. Die zur Laufzeit erzeugten Objekte werden als Ausprägungen einer Klasse bezeichnet. Die neue Farbe, die die Marken der zusammengefalteten Objektnetze unterscheidet, ist die Objekt-ID. Innerhalb des Klassennetzes werden alle Stellen und Kanten durch die Objekt-ID ergänzt. Dadurch ist gewährleistet, daß die zusammengefalteten Objekte eindeutig unterschieden werden können. Die IDs der jeweiligen Objekte werden lokal aufbewahrt und können von den Methoden des Klassenobjekts benutzt werden. Die ID eines neu erzeugten Objekts muß eindeutig sein. Das ist gewährleistet, wenn z.B. ein Zählerwert innerhalb dieser Klasse benutzt wird, der bei jedem Erzeugen einer neuen Objekt-ID inkrementiert wird. Die OID besteht dann aus der Kombination von Klassenname und Zählerwert.

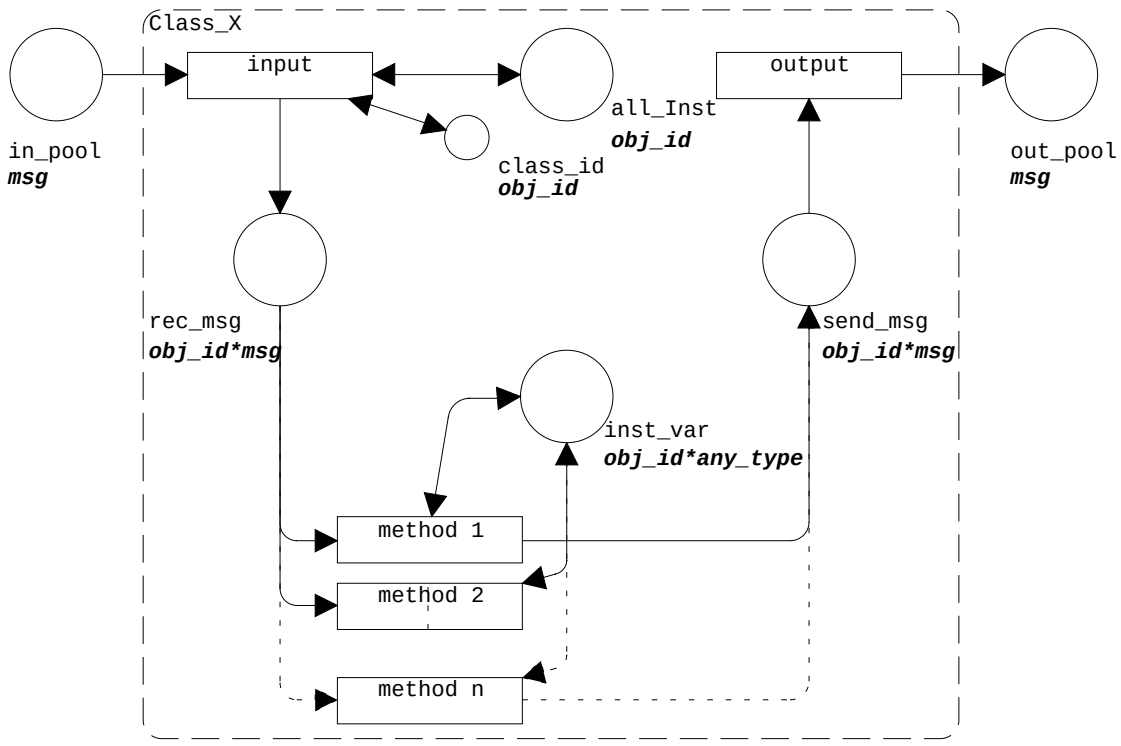


Abbildung 85: Klassennetz nach [Moldt 96]

Ein Objekt wird in [Moldt 96] durch den Aufruf der `new`-Methode eines Klassennetzes erzeugt. Diese ist in Abbildung 86 dargestellt. Bei der Erzeugung eines Objekts werden dessen Attribut-Stellen initialisiert. Technisch wird kein neues Netz angelegt, sondern, wie oben beschrieben, eine neue Objekt-Identifikation (OID) im Klassennetz generiert und diese dann mit den Attributen kombiniert. Die `new`-Methode ergänzt die gefalteten Objektnetze. Sie wird aufgerufen, indem eine Nachricht an das Klassenobjekt geschickt wird. Das Klassennetz hat die ID '(classname, 0)'. Die input-Transitionen aller Objekte einer Klasse werden zu der Transition `input-class` zusammengefaltet. Bei der in eckigen Klammern eingeschlossenen Beschriftung der Transiton handelt es sich um einen Guard, der in der funktionalen Programmiersprache ML kodiert ist. Der Guard überprüft in diesem Fall, ob die Empfänger-ID in der Nachricht der Klassen-ID in der Stelle `class_id` entspricht. Wenn dies der Fall ist, wird die Nachricht in die Stelle `rec_msg` weitergeleitet. Die `new`-Methode erzeugt eine neue ID für das neue Objekt und initialisiert die Instanzvariablen. Für die Vernichtung von Objekten werden in [Moldt 96] zwei Protokolle skizziert, auf die hier nicht weiter eingegangen wird.

Objekte unterscheiden sich neben der OID durch ihren Zustand. Dieser definiert sich durch die Markierung der Objektnetze. Diese setzt sich zusammen aus der Belegung der Attributstellen und durch den Zustand der objektinternen Steuerung, d.h. den Zustand der Methodenausführung. Der Systemzustand definiert sich aus den Objektzuständen, den Klassenzuständen und den Nachrichten.

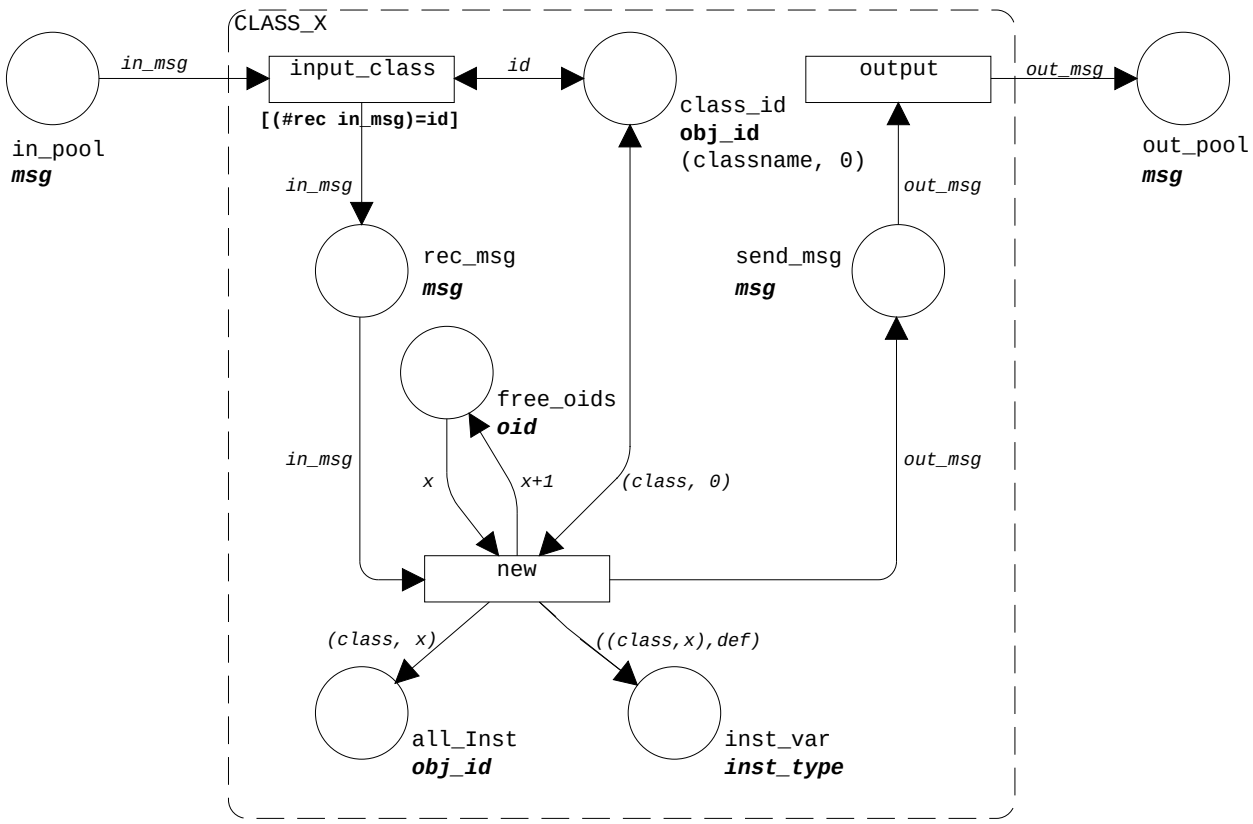


Abbildung 86: New-Methode des Klassennetzes

4.2.3 Objektnetze unter Verwendung von Interfaces

Das Konzept der aus der objektorientierten Programmierung bekannten Interfaces wird bei einem anderen Ansatz zur Modellierung von objektorientierten Petrinetzen verwendet. Ein Interface definiert einen Typ, der spezifiziert, welche Nachrichten ein Objekt von diesem Typ verarbeiten kann, d.h. die Methoden mit ihren Signaturen. Interfaces implementieren keine Methode, so daß es von einem Interface keine Ausprägungen gibt. Ein Interface kann von einer oder mehreren Klassen implementiert werden und eine Klasse kann mehrere Interfaces implementieren. Um ein Interface zu implementieren, muß die Klasse für jede im Interface angegebene Methode eine Imlementation liefern. Das Interface-Konzept dient der Sauberkeit bei der Modellierung durch klar definierte Schnittstellen, ermöglicht die Verwendung von Mehrfachvererbung durch die Möglichkeit, daß eine Klasse mehrere Interfaces implementiert und steigert die Wiederverwendbarkeit von Klassen. Durch die Verwendung von Interfaces gibt es eine klare Trennung zwischen der Beschreibung von Klassen und deren Implementation. In der Abbildung 87 ist die Verwendung eines Interfaces in UML-Notation aus [Fowler 97] dargestellt. Die Klasse ‘I’ ist ein Interface, das von der Klasse ‘A’ implementiert wird. Eine ausführliche Diskussion der Interface-Konzepte findet man in [Cornell 97].

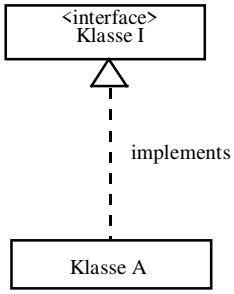


Abbildung 87: Verwendung eines Interface

Die Abbildung 88 zeigt ein Interface in schematischer Netzdarstellung. Es definiert die Signaturen der Methoden mit ihren Parametern und deren Typen. Die Selektoren für die definierten Methoden sind durch die mit ‘m1’ bis ‘m3’ beschrifteten Transitionen dargestellt. Die grafisch hervorgehobenen Stellen repräsentieren den Kontrollfluß. Die Marken in diesen Stellen zeigen den Ausführungszustand des Netzes an. Die Farben der anderen Ausgangsstellen entsprechen den Typen der Parameter. Die Methode ‘m1’ hat zwei Parameter vom Typ Integer, die Methode ‘m2’ drei Parameter vom Typ String und ‘m3’ hat keine Parameter. Die Kantenbeschriftungen ‘m1’ bis ‘m3’ entsprechen den Nachrichten für die Methoden, die durch die, hier gleichnamigen, Methoden-Selektoren spezifiziert werden. Auf der rechten Seite wird der Rückgabetyp der Methoden definiert. In diesem Beispiel liefert die Methode ‘m1’ einen Integerwert zurück, ‘m3’ einen Wert vom Typ Boolean und die Methode ‘m2’ liefert keinen Wert zurück. In der Stelle *Call* werden alle Nachrichten an Objekte abgelegt. Durch eine Marke in der Nebenstelle *Class* und einen hier nicht explizit aufgeführten Guard, werden die Nachrichten an Objekte, die dieses Interface implementieren, von den Transitionen ‘m1’ bis ‘m3’ selektiert. Bei den ungerichteten Kanten handelt es sich um Testkanten, wie in [Christensen und Hansen 93] beschrieben.

Die eigentliche Implementation der Methoden verwenden die auf den Parameterstellen zur Verfügung gestellten Marken zur Berechnung. Die Methoden sind auf der Klassen-Seite realisiert. Dies ist in der Abbildung 89 dargestellt. Durch Fusions-Stellen wird die Interface-Seite aus der Abbildung 88 mit der Klassen-Seite verbunden. Sofern es einen Rückgabewert gibt, wird dieser als Marke auf eine Ausgangsstelle der Methode gelegt. Vom Interface wird dieser Wert als Parameter in eine Nachricht eingebettet und diese auf die Stelle *Return* gelegt. Das aufrufende Objekt erhält dann diese Antwortnachricht.

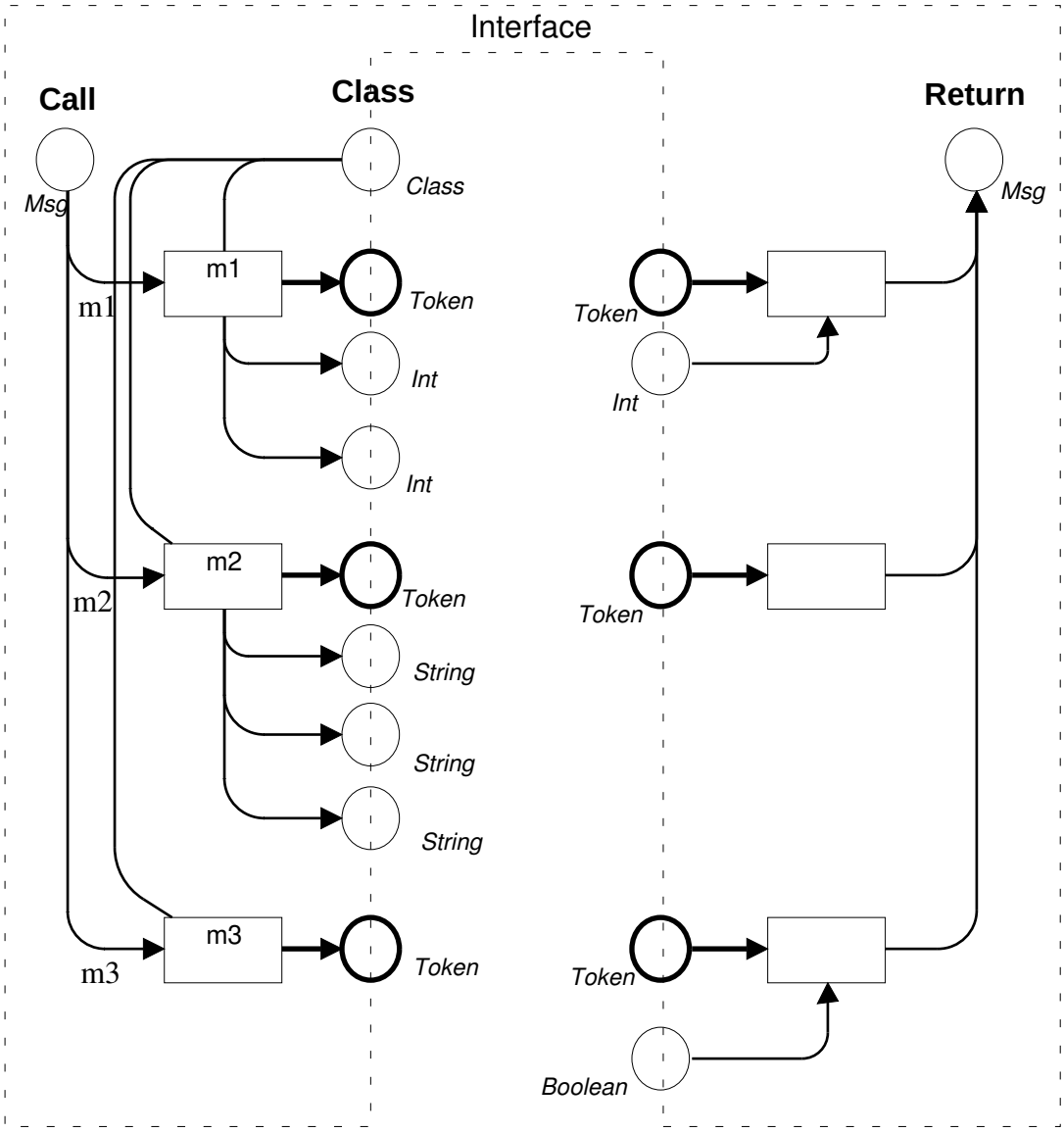


Abbildung 88: Interface

Bei dem hier vorgestellten Interface-Konzept werden die Nachrichten im Interface in ihre Komponenten zerlegt. Das hat den Vorteil, daß diese Aufgabe nicht von den Methodenimplementationen ausgeführt zu werden braucht. Für die Zerlegung könnte auch eine, für alle Nachrichtenzerlegungen gemeinsame Unterseite verwendet werden. Diese gemeinsame Unterseite wird Methodenseite genannt und später vorgestellt.

In der Abbildung 89 ist die Klasse dargestellt, die das Interface aus Abbildung 88 implementiert. Diese Klasse verwendet das Interface als Unterseite. Diese Architektur hat den Grund, daß bei der verwendeten Software Design/CPN nur Unterseiten mehrfach wiederverwendet werden können und es bei Interfaces möglich sein soll, diese durch mehrere Klassen zu implementieren. Die Transitionen ‘meth1’ bis ‘meth3’ implementieren die vom Interface definierten Methoden. In dieser Abbildung sieht man, daß die Methoden die ihren Parametern und dem Kontrollfluß entsprechenden Marken aus den Eingangsstellen abziehen und Marken als Ergebnis und eine Kontrollflußmarke in die Ausgangsstellen legen.

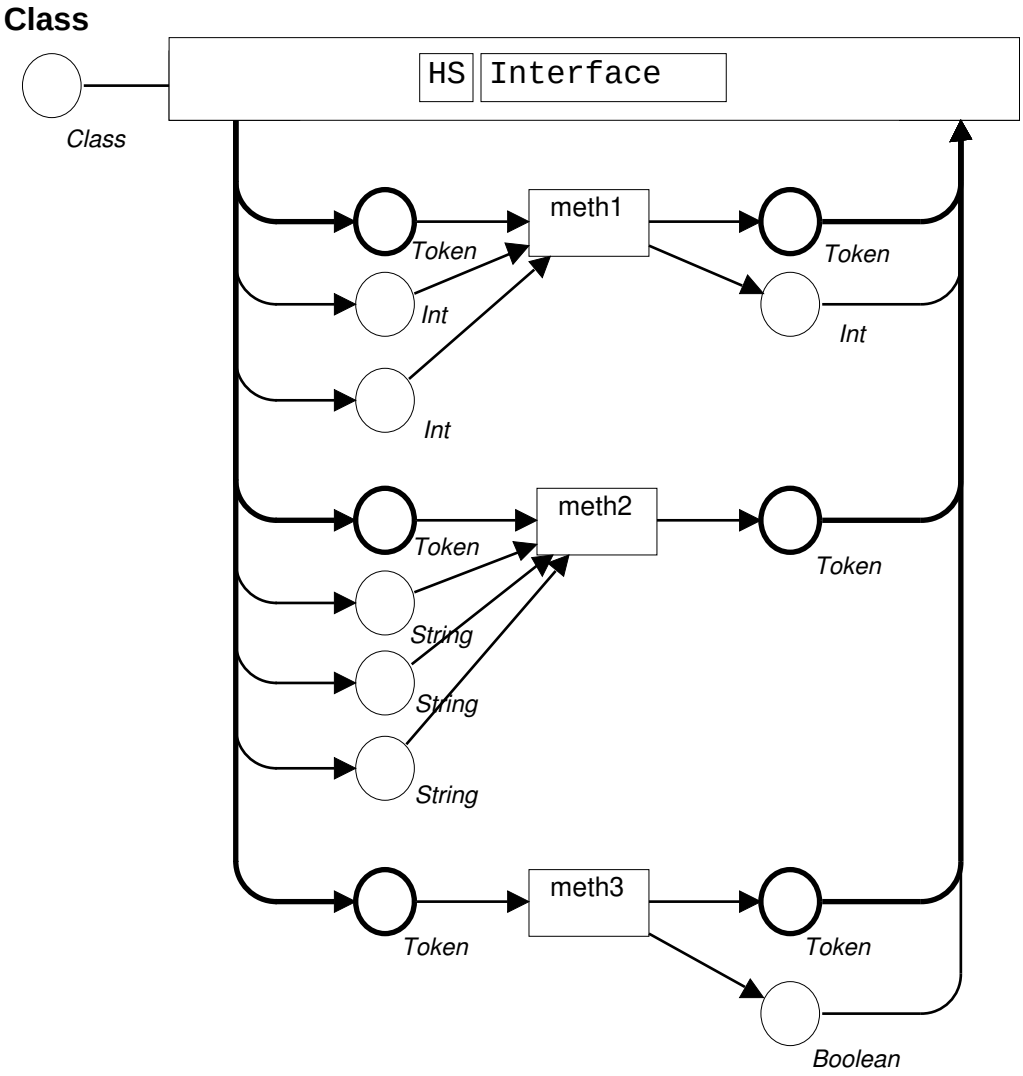


Abbildung 89: Implementation des Interface

Auf der Hierarchieseite in Abbildung 90 sieht man die hierarchische Struktur eines Petrinetzes bei der Verwendung eines Interface. Die Klasse ‘Class’ implementiert das Interface. Ein Interface kann von mehreren Klassen implementiert werden. Für jede implementierende Klasse gibt es eine Kopie der Interface-Seite. Die Seiten ‘Method1’ bis ‘Method3’ sind Unterseiten der Seite ‘Class’ und enthalten die Implementationen der vom Interface definierten Methoden. Bei der Verwendung von Interfaces zur Modellierung von OOCPN brauchen die Interfaces nur einmal gezeichnet zu werden.

Die Methoden werden auf Unterseiten spezifiziert. Die Funktionalität wird i.a. durch Kantenbeschriftungen bzw. Code-Regionen festgelegt. Der Kontrollfluß kann durch die Verwendung der Kontrollkonstrukte aus der Abbildung 91 modelliert werden. Durch das fork-Konstrukt wird der Kontrollfluß in nebenläufige Ausführungspfade aufgeteilt. Durch das join-Konstrukt werden nebenläufige Ausführungspfade zusammengeführt. Zur Modellierung von booleschen Bedingungen werden in der Abbildung 91 zwei Varianten vorgestellt. Bei der oberen Darstellung fällt der Kontrollfluß mit dem Datenfluß zusammen, in der unteren Darstellung wird dieser explizit modelliert. Semantisch besteht kein Unterschied. In [v.d. Aalst 97] werden diese Kontrollkonstrukte unter den Namen And-Split, And-Join, Or-Split und Or-Join verwendet. Im drittenTeil des Anhangs sind weitere Kontrollkonstrukte abgebildet.

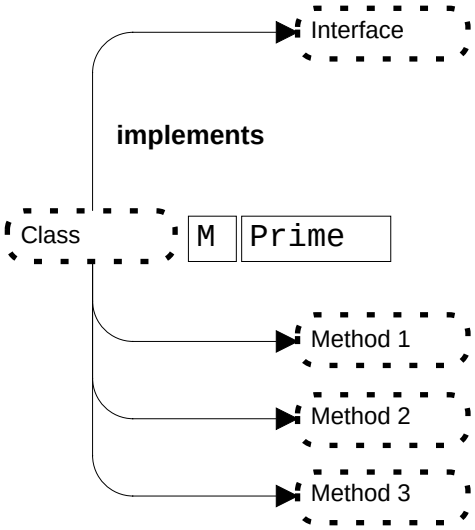


Abbildung 90: Hierarchieseite

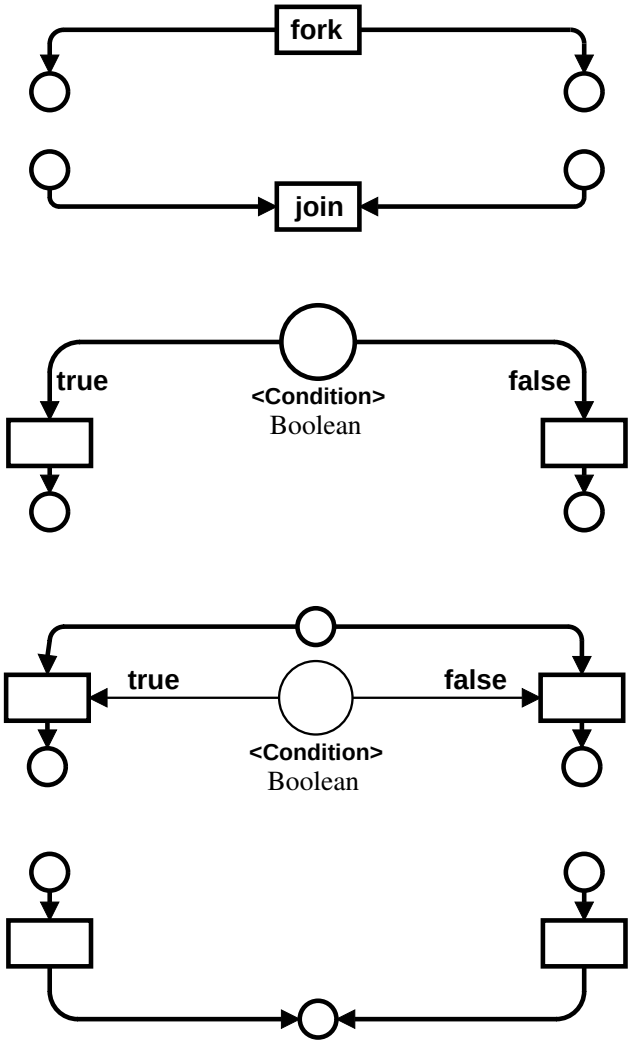


Abbildung 91: Kontrollfluß-Konstrukte

4.2.4 Klassennetz beim Interface-Ansatz

Auch bei dem Interface-Ansatz werden Objektnetze zusammengefaltet. Wie in [Moldt 96] unterscheiden sich auch hier die Objekte durch eine eindeutige Objekt-ID. Der Farbtyp der Instanzvariablen muß bei der Faltung um die OID erweitert werden. Wird eine new-Methode zur Erzeugung eines Objekts aufgerufen, so wird eine Nachricht an die Klasse geschickt, die den Typ dieses Objekts definiert. Während in [Moldt 96] bei einem Aufruf einer new-Methode einer Klasse die Attribute des neu erzeugten Objekts sofort instanziiert werden, wird bei diesem Ansatz zunächst nur eine neue OID für ein neues Objekt erzeugt. Nun kann die new-Methode des neu erzeugten Objekts zur Initialisierung der Attribute bezüglich dieser neuen Objekt-ID aufgerufen werden. Dieser Ansatz entspricht dem Konzept der Objekterzeugung in der Programmiersprache Java ([Cornell 97]), bei dem neue Ausprägungen von Klassen in einem sogenannten Konstruktor, oben als new-Methode bezeichnet, verwendet werden, um Objekte zu initialisieren.

Dargestellt ist die Erzeugung einer neuen Objekt-ID in der Abbildung 92. Dieser Mechanismus befindet sich auf einer Unterseite und wird von allen Klassen verwendet, wenn eine new-Methode aufgerufen wird. Das ‘C’ bezeichnet eine Nachricht an eine Klasse, nämlich an die Klasse mit dem Namen ‘class’. Aufgerufen wird dort die Methode (M) ‘new’. Der letzte Teil ‘msg’ definiert die Parameter, die an die Initialisierungsmethode des neu erzeugten Objekts weitergereicht werden. Die Transition erzeugt eine neue OID für das neu erzeugte Objekt, bestehend aus dem Klassennamen und einer eindeutigen Nummer (oid) und sendet an dieses Objekt eine Nachricht mit dem Aufruf der new-Methode zur Initialisierung des Objekts. Das Empfängerobjekt wird durch ‘I(class,oid)’ bezeichnet. Dabei steht das ‘I’ für ‘instance’.

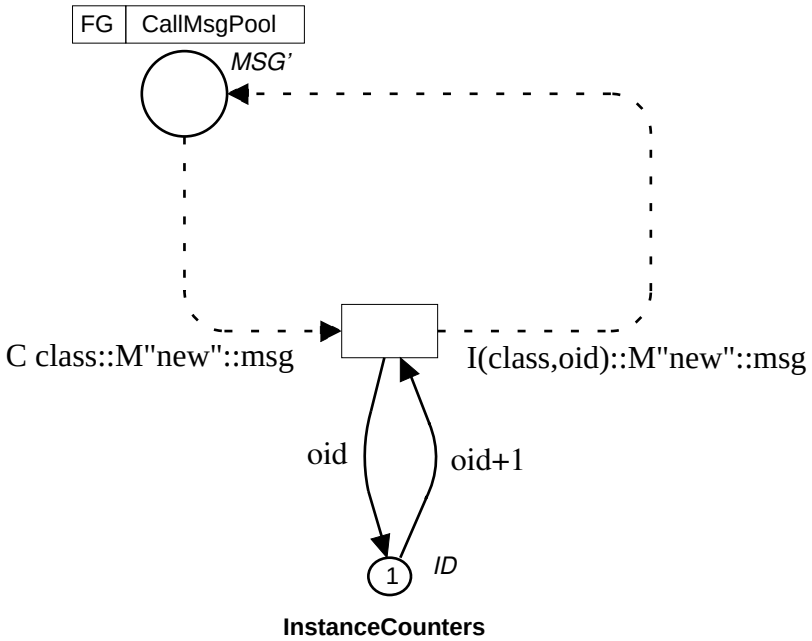


Abbildung 92: Erzeugen einer ObjektID

Wird statt der Interfaces die von allen Methoden gemeinsam genutzte Methoden-Seite verwendet, so sieht eine new-Methode so aus, wie in der Abbildung 93. Die hier gezeigte new-Methode stammt aus einem Beispiel zur Erzeugung von Listen. Die Beschriftung ‘HS’ in der oberen Transition deutet an, daß auf einer Unterseite eine Verfeinerung dieser Transition realisiert ist. Bei dieser Unterseite handelt es sich um die Methoden-Seite, die in einem nachfolgenden Abschnitt ausführlich erklärt wird. Sie zerlegt die Nachrichten in ihre Bestandteile und stellt damit den Methoden die benötigten Para-

meter zur Verfügung. In diesem Falle sind dies die Marken ‘List’ und ‘Pair’ in den Eingangsstellen. Die Realisierung der Methoden-Seite als Unterseite ist auf der zur Modellierung verwendeten Software ‘Design/CPN’ begründet. Diese Seite kann so von vielen Methoden verwendet werden. Ist die untere Transition aus der Abbildung 93 aktiviert, so kann sie schalten und legt eine neue Marke vom Typ ‘cons’ auf die Stelle ‘Pair’, bei der es sich um die Attributstelle des Objekts handelt. Die Stelle ‘this’ vom Typ ‘List’ enthält die OID des gerade aufgerufenen Objekts und wird benötigt, um die Instanzvariable ‘cons’ mit der korrekten “Färbung“ abzulegen.

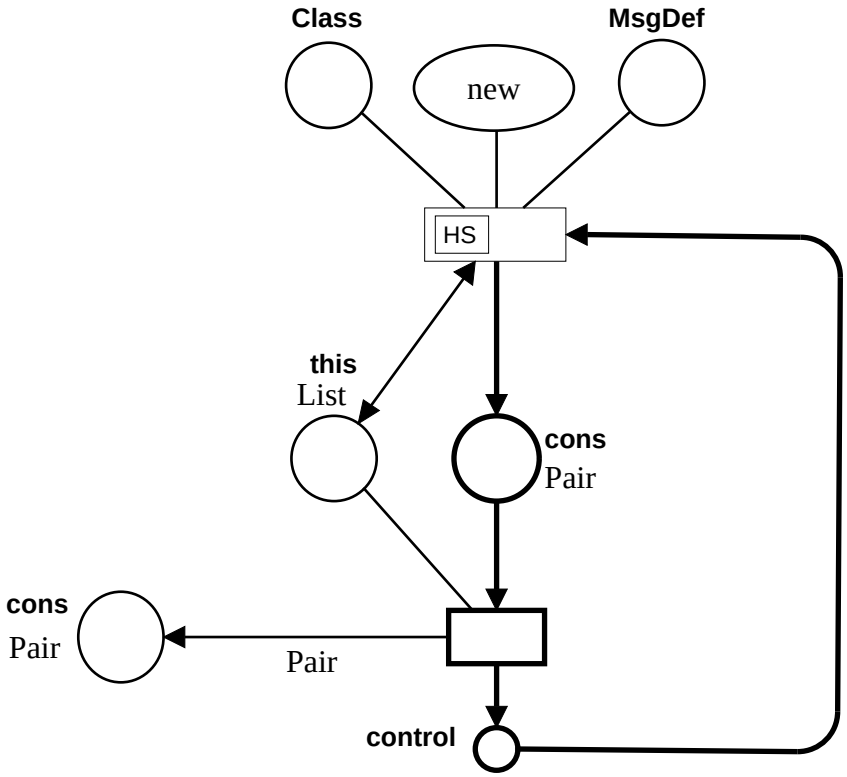


Abbildung 93: new-Methode zur Erzeugung einer Liste

4.3 Nachrichtenformat und Methodenaufruf

Für das Nachrichtenformat werden zwei verschiedene Ansätze vorgestellt. Sie unterscheiden sich in der Anzahl der Bestandteile der Nachrichten und durch die unterschiedliche Verwendung der Formate bei der Kommunikation zwischen Objekten. Weiterhin werden unterschiedliche Realisierungen des Methodenaufrufes in diesem Kapitel diskutiert. Zum Abschluß erfolgt eine Betrachtung der hierarchischen Struktur der Netze.

4.3.1 Nachrichtenformat zur asynchronen Kommunikation

Nachrichten, die zwischen Objekten ausgetauscht werden, sind in [Moldt 96] als Marken dargestellt. Nachrichten bestehen aus einer Nachrichtennummer, einem Sender, einem Empfänger, der rufenden Methode, der aufgerufenen Methode, einer Selbstreferenz, einer Parameterliste und einer Nummer für Antwortnachrichten. Die Nachrichten, die verschickt werden, besitzen alle den gleichen Aufbau. Genauere Angaben zu diesen Bestandteilen einer Nachricht findet man in der folgenden Aufzählung. Dabei ist eine Nachricht ein Tupel mit den Feldern:

- *msg_no:* Eindeutige Nummer, die von dem sendenden Objekt für die Nachricht vergeben wird
- *send:* Objekt-ID des sendenden Objekts
- *rec:* Objekt-ID des empfangenden (gerufenen) Objekts
- *c_meth:* Name der rufenden Methode im Objekt *send*
- *meth:* Name der gerufenen Methode im Objekt *rec*
- *c_self:* Objekt-ID des Objekts, das bei einer Selbst-Referenz gerufen werden soll
- *arg:* Parameterliste der aufgerufenen Methode, bzw. der Rückgabewert bei Antwortnachrichten
- *rep_no:* Diese Nummer gibt bei Antwortnachrichten an, zu welcher Nachricht diese Nachricht die Antwort ist. Dadurch ordnet das Objekt, das den Methodenaufruf initiiert hat, die Antwortnachricht dem Ergebnis seines Methodenaufrufs zu.

Nachrichten werden in [Moldt 96], wie in Abbildung 94 gezeigt, von Methoden empfangen, verarbeitet und es wird eine Antwortnachricht erstellt. Die Methode `method` berechnet z.B. den Wert `f(x)` in Abhängigkeit von der Instanzvariablen `x`. Diese Berechnung kann in einer Verfeinerung der Transition realisiert sein. Die Nachrichtennummer `num` wird einem lokalen Pool mit eindeutigen Nummern entnommen, der hier durch die Stelle `send_pool` dargestellt wird. Mit `self` wird die eigene Objekt-ID bezeichnet. Die Objekt-ID des Empfängers wird der ursprünglichen Nachricht `in_msg` entnommen. Die jetzige rufende Methode `c_meth` ist die Methode `method`. Die rufende Methode in der `in_msg` ist bei der Antwortnachricht die Zielmethode `meth`. Eine eventuelle Selbstreferenz, die nur im Falle von Codevererbung eine Bedeutung hat, durch (0,0) gekennzeichnet, würde an das Objekt selbst gehen. Der Rückgabewert `arg` der Methode ist das Ergebnis der Berechnung. Die Antwortnummer `rep_no` übernimmt den Wert der `msg_no` der Nachricht `in_msg`. Der Nachrichtenaufbau erfordert, daß eine Nachricht von jeder Methode zerlegt und wieder zusammengebaut wird. Ein weiterer Aspekt ist, daß der Nachrichtenaufbau immer gleich ist. Es wird nicht zwischen Aufrufnachrichten und Antwortnachrichten unterschieden. Da ein Bezug auf vorhergehende Nachrichten möglich sein soll, ist das Nachrichtenformat relativ aufwendig.

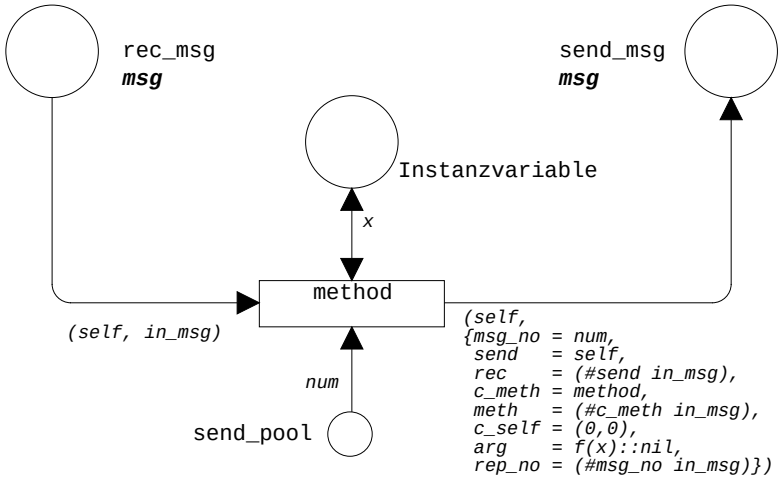


Abbildung 94: Erstellen der Antwortnachricht durch eine Methode in [Moldt 96]

4.3.2 Nachrichtenformat zur synchronen Kommunikation

Eine Weiterentwicklung ist der folgende Ansatz von Kummer, Moldt und Wienberg in [KMW 98]. Dort wird das Nachrichtenformat vereinfacht, indem es auf die in der Programmierung benutzte Aufrufsemantik spezialisiert wird. Nachrichten werden als Tupel aus folgenden Komponenten definiert: einem Empfängerobjekt, dessen Methode aufgerufen wird, einem Senderobjekt, das auf eine Antwort wartet, einem Methodennamen und einer Parameterliste. Es wird zwischen einer call message und einer return message unterschieden. Der Aufbau einer return message ist einfacher als der einer call message, weil dort der Sender und der Methodenname nicht angegeben werden müssen. Wichtig sind in dem Fall nur der Empfänger (das ursprünglich sendende Objekt der call message) und die Parameterliste, üblicherweise ein einzelner Wert.

Der Aufbau einer call message (CM) sieht in [KMW 98] wie folgt aus:

CM= [Methodenname, Empfängerobjekt, Senderobjekt, Param1, Param2, Param3, ...]

Der Aufbau einer return message (RM) ist dagegen einfacher:

RM= [Empfängerobjekt, Ergebnis]

Das Ergebnis bei der return message ist optional.

Die Erstellung der Nachricht erfolgt in [KMW 98], wie in Abbildung 95 gezeigt, mittels einer Transition. Die Nachricht wird in dieser Transition, aus den verschiedenen oben beschriebenen Komponenten, d.h. Methodenname, Empfänger, Sender und Parameter, die als Marken auf den entsprechenden Stellen liegen, aufgebaut. Die *callid* wird durch einen globalen Zähler erzeugt und ist somit eindeutig. Diese *callid* wird in der *Wait* Stelle verwendet um die Antwortnachricht eindeutig zuzuordnen. Von den Stellen werden nur die Marken von der Kontroll-Stelle und der SenderObjekt-Stelle wirklich abgezogen. Die anderen Marken verbleiben auf den Stellen und stehen für weitere Methodenaufrufe zur Verfügung. Somit können Methoden eines Objekts mehrfach, nebenläufig, aufgerufen werden.

Die Stelle *Kontroll* zeigt die aktivierten Transitionen und damit den Zustand des Netzes an. Diese Stelle und die zugehörenden Kanten werden durch stärkere Linien hervorgehoben, während die Daten-Stellen und die dazugehörenden Kanten durch normale Linienstärke dargestellt werden. Das in der Abbildung 95 dargestellte Netz ist Teil der zur Kommunikation zwischen Objekten verwendeten Call-Seite. Diese wird an anderer Stelle ausführlich beschrieben. Hier wird nur die Nachrichtenkonstruktion betrachtet.

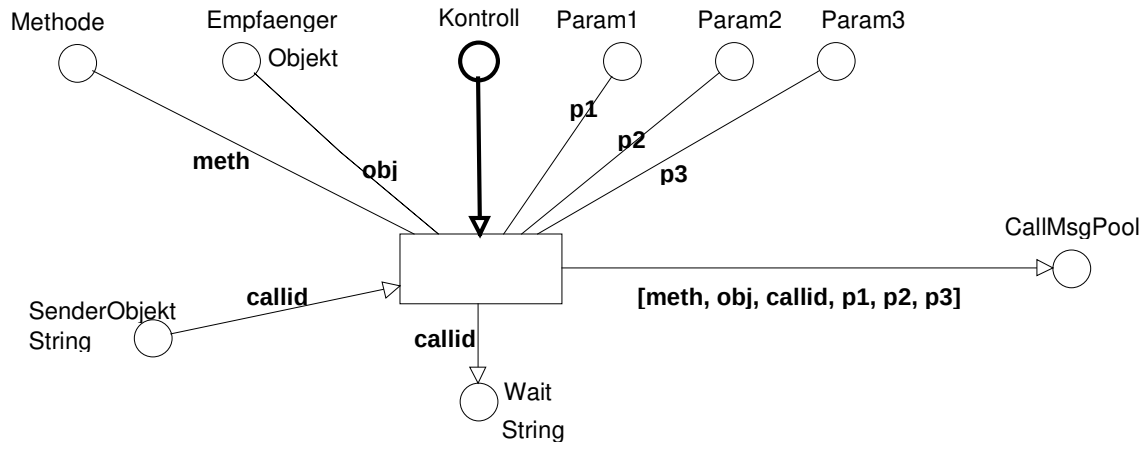


Abbildung 95: Nachrichtenzusammenbau nach [KMW 98]

Die Unterschiede zwischen den Ansätzen aus [Moldt 96] und [KMW 98] sind:

- Der Aufbau der Nachrichten vereinfacht sich in [KMW 98] gegenüber [Moldt 96] aufgrund der Unterscheidung zwischen call und return messages.
- In [KMW 98] gehört zu jeder call-message eine return-message. Diese kann durch die Angabe des SenderObjekts eindeutig zugeordnet werden. Als Methodenaufruf wird ein Call mit dem zugehörigen Return verstanden. Dadurch vereinfacht sich die Nachrichtenverwaltung und es wird sichergestellt, daß keine Nachricht unbeantwortet bleibt.
- Für jeden Methodenaufruf wird nach [KMW 98] eine neue Netzinstanz erzeugt. Dies ist bei dem Ansatz von [Moldt 96] nicht der Fall. Dort wird das zu sich selbst nebenläufige Schalten einer Methode durch die erwähnte Nachrichtennummer in der Nachricht realisiert.
- Die Verarbeitung der Nachrichten durch die Methoden vereinfacht sich in [KMW 98] dadurch, daß die Nachrichten in ihre Bestandteile zerlegt werden, bevor diese dann von der Methode zur Verarbeitung verwendet werden. Dieser Mechanismus wird weiter unten erklärt.

4.3.3 Nachrichtenzerlegung in der Methode

In früheren Arbeiten von [Moldt 96] und [Maier 97] erfolgte der Methodenaufruf über einfachen Nachrichtenverkehr. Die Nachrichten wurden von Objekten empfangen, die benötigten Bestandteile durch die Methode aus der Nachricht entnommen, die entsprechende Methode ausgeführt und dann eine Antwortnachricht aus einzelnen Komponenten generiert und versendet. In der Abbildung 96 ist eine Methode dargestellt, deren Methodenname in einer Stelle als Nebenbedingung abgelegt ist und die auf die Instanzvariablen des Objekts zugreift.

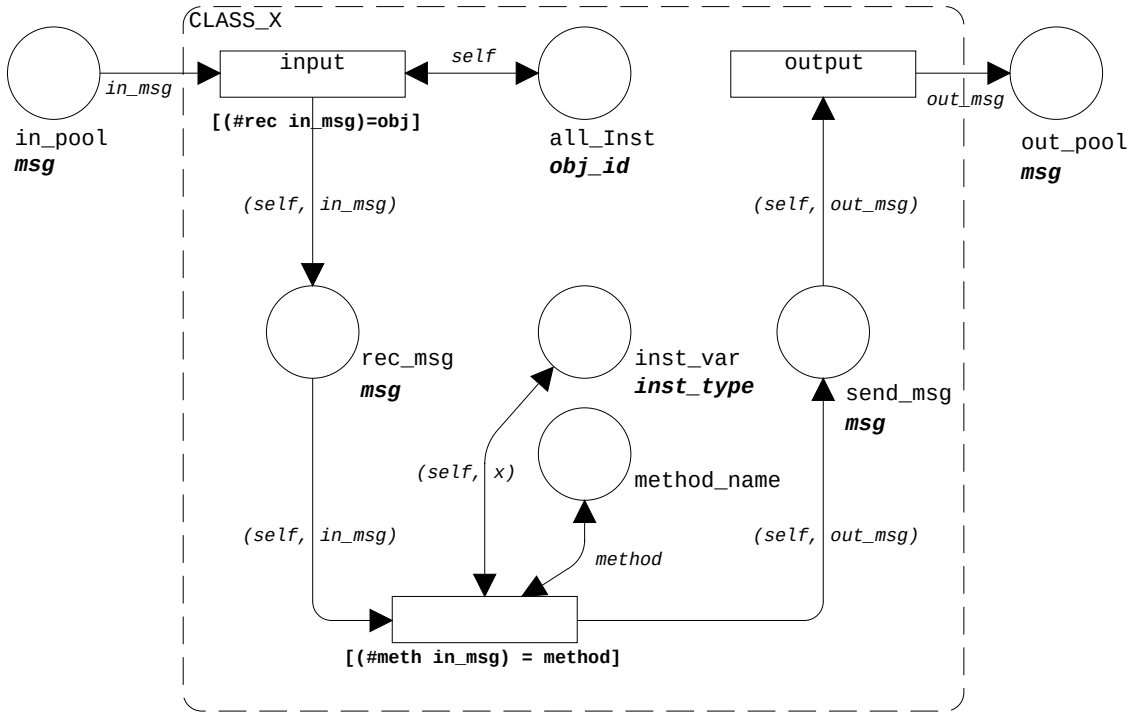


Abbildung 96: Methodenaufruf aus [Moldt 96]

4.3.4 Nachrichtenzerlegung vor der Methodenausführung

Prinzipiell erfolgt der Methodenaufruf in [KMW 98] wie in Abbildung 97 dargestellt. Unterschiede zu [Moldt 96] bestehen darin, daß die Nachricht grundsätzlich zuerst in ihre Bestandteile zerlegt und diese dann der Methode zur Ausführung übergeben werden. Bei diesem Ansatz geschieht dies jedoch nicht durch Interfaces sondern durch eine spezielle Methoden-Seite wie in Abbildung 98. Nachdem die Methode abgearbeitet wurde, wird die Antwortnachricht von einer anderen Transition konstruiert und zurückgeschickt. Die Methoden-Seite ist so konstruiert, daß sie bei verschieden Methodenaufrufen immer wieder verwendet werden kann. Sie dient also zur Aufbereitung der Nachrichten für die Methoden und für die Erzeugung einer Antwortnachricht. Diese Antwortnachricht wird von einer speziell für die Kommunikation benötigten Seite, der Call-Seite, der ursprünglichen Nachricht zugeordnet. Die Funktionsweise der Call-Seite wird im folgenden Abschnitt erklärt.

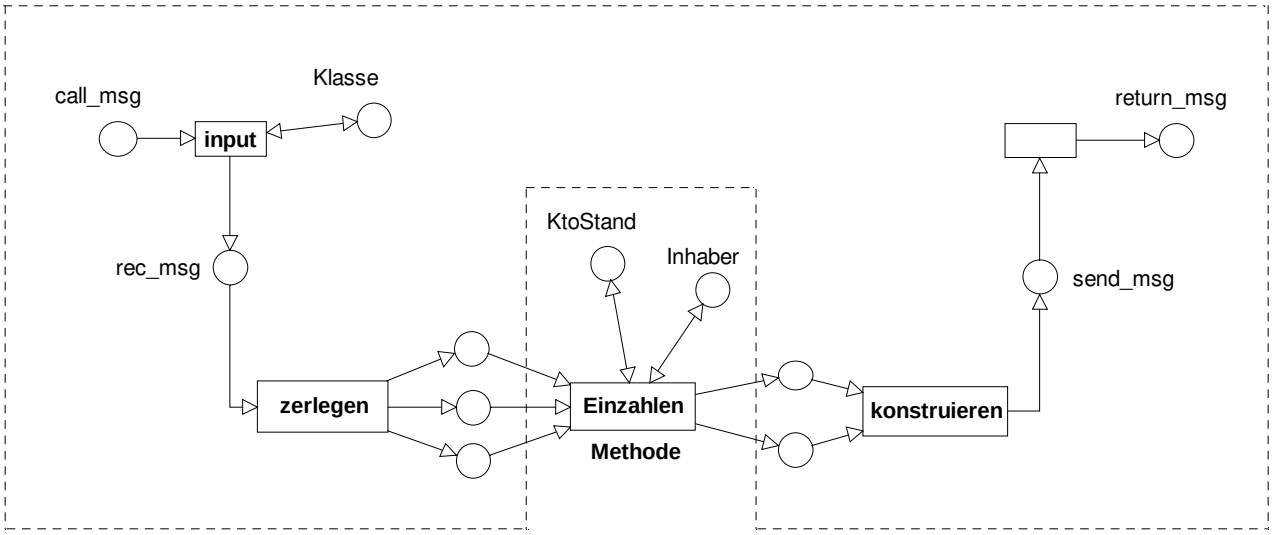


Abbildung 97: Prinzip des Methodenaufrufs in [KMW 98]

Abbildung 98 zeigt diese in [KMW 98] für den Methodenaufruf definierte Methoden-Seite. Diese nimmt die gesendete Nachricht in Empfang und teilt sie in ihre Bestandteile auf, wobei jede Komponente in einer bestimmten Stelle abgelegt wird. Durch die Marke in der Kontroll-Stelle wird angegeben, in welcher Ausführungsphase sich das Netz befindet. Die maximale Anzahl von Parametern wird in diesem Netz auf drei beschränkt. Die maximale Anzahl von Parametern muß beim Netzentwurf festgelegt werden. Die aufgerufene Methode arbeitet mit den gelieferten Daten und gibt ihr Ergebnis zurück. Der Aufrufer wird gespeichert, um nach der Durchführung der Methode Bestandteil der Antwortnachricht zu werden. Der Vorteil dieser Konstruktion ist die Möglichkeit der Wiederverwendung der Methoden-Seite, so daß die Methode selbst nicht durch das Zerlegen und Zusammensetzen der Nachrichten belastet wird. Die Funktionalität des Zerlegens von Nachrichten in ihre Bestandteile und die Konstruktion von Antwortnachrichten ist die gleiche, wie bei den Interfaces, die Methoden-Seite kann aber nicht zur Realisierung von Interface-Konzepten verwendet werden. Ein Nachteil bei der Verwendung der Methoden-Seite ist die Beschränkung bezüglich der Zahl der möglichen Parameter. Ein konkretes Beispiel für einen Methodenaufruf wird am Ende des folgenden Abschnitts behandelt. Dort werden die hier vorgestellten Prinzipien angewendet.

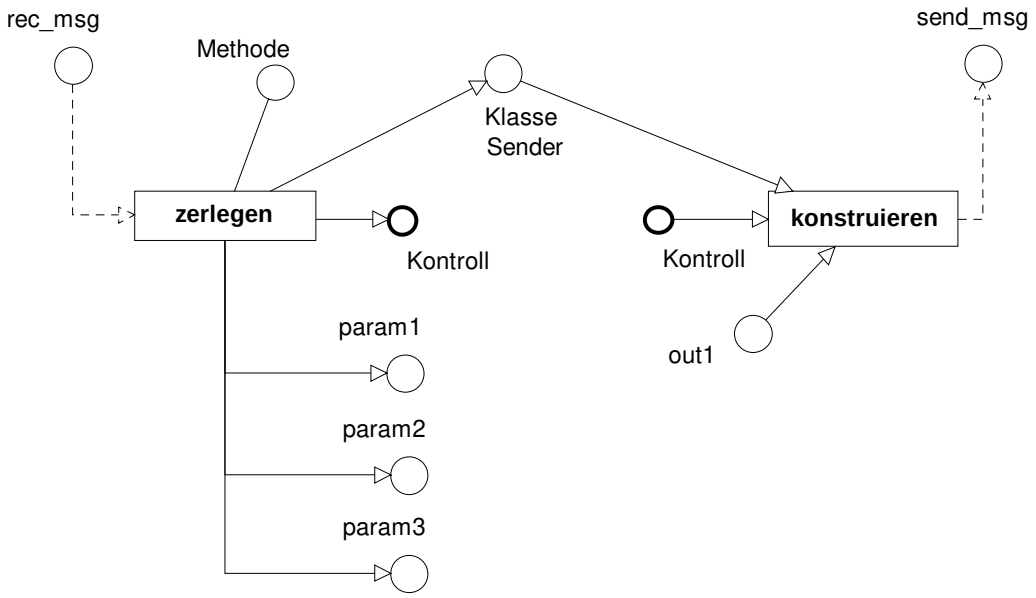


Abbildung 98: Methoden-Seite

4.3.5 Hierarchische Struktur

Die Abbildung 99 zeigt einen Ausschnitt aus der Hierarchie-Seite eines Beispiels, in dem Objekte vom Typ List und Pair erzeugt werden können. Die Pfeile zeigen zu den Seiten, die in tieferen Ebenen der Hierarchie liegen. Wie daraus zu ersehen ist, liegt u.a. die Methoden-Seite ‘Method#5’ (die 5 ist die Seitennummer) unten in der Hierarchie und wird von allen Objektmethoden, die als Oberseiten modelliert werden, aufgerufen. Für jede Methode wird eine neue Instanz der Methoden-Seite erzeugt. Die oberen Seiten, hier die Seiten ‘Pair#14’ und ‘List#15’, entsprechen den Objekten mit ihren Attributen und Methoden. Die Verfeinerung der Methoden ist auf den darunterliegenden Seiten modelliert. Die Definition der obersten Seiten als Primepages ist Design/CPN bedingt und wird zur Simulation benötigt, damit von diesen Seiten zu Beginn der Simulation Instanzen erzeugt werden.

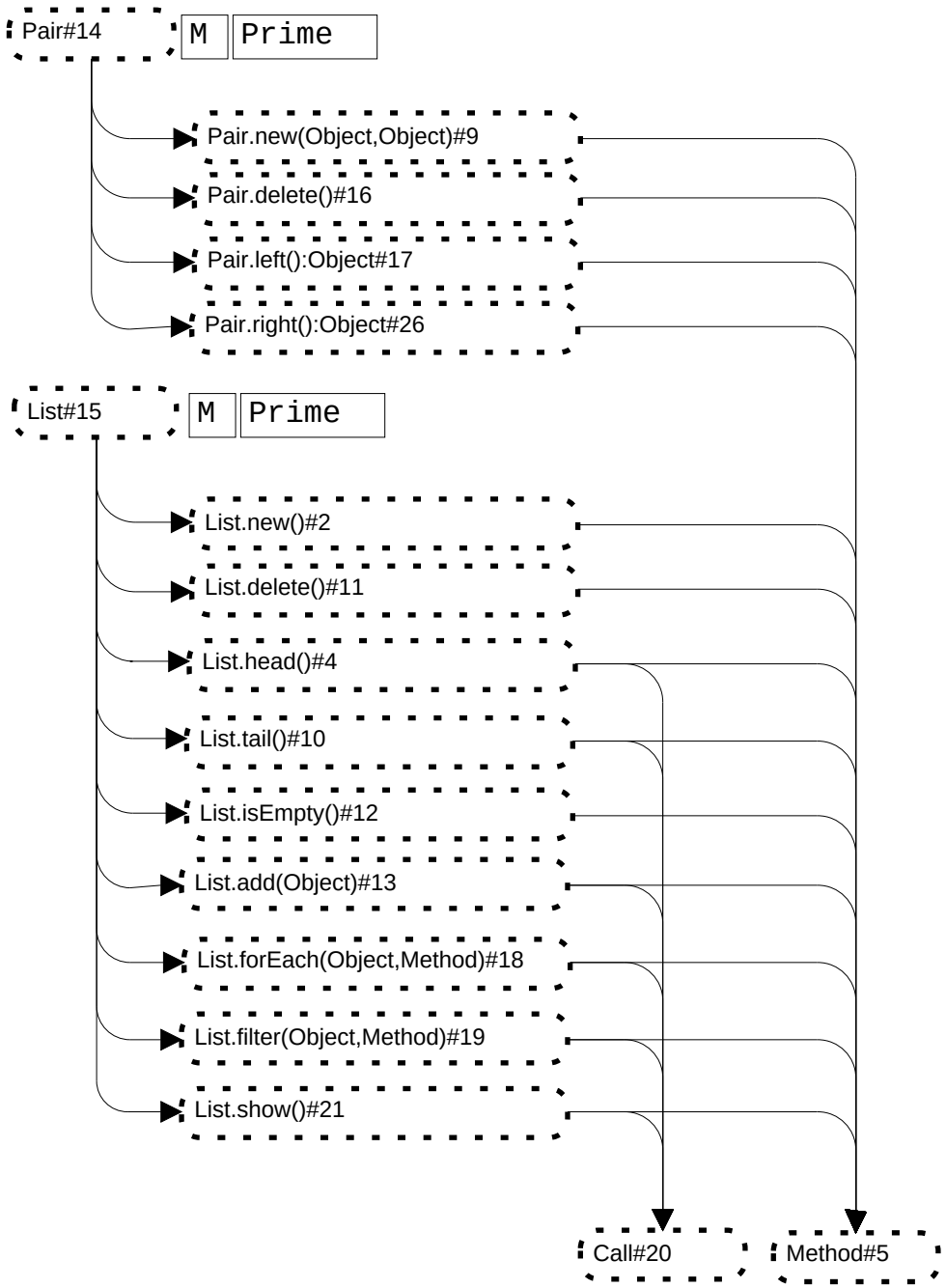


Abbildung 99: Hierarchie-Seite

In der Abbildung 100 ist die Verfeinerung einer Transition dargestellt. Dieses Konzept wird häufig verwendet, um übersichtliche Netze zu erzeugen und um von bestimmten Details zu abstrahieren. Innerhalb der Transitionsverfeinerung können beliebig komplexe Netze mit nebenläufig schaltenden Transitionen modelliert und in der Vergröberung als Blackbox betrachtet werden. In Abbildung 100 ist der Transition auf der linken Seite nur zu entnehmen, daß sie drei Eingangsstellen und eine Ausgangsstelle hat. In der Verfeinerung auf der rechten Seite ist zu erkennen, in welcher Reihenfolge die Marken abgezogen werden.

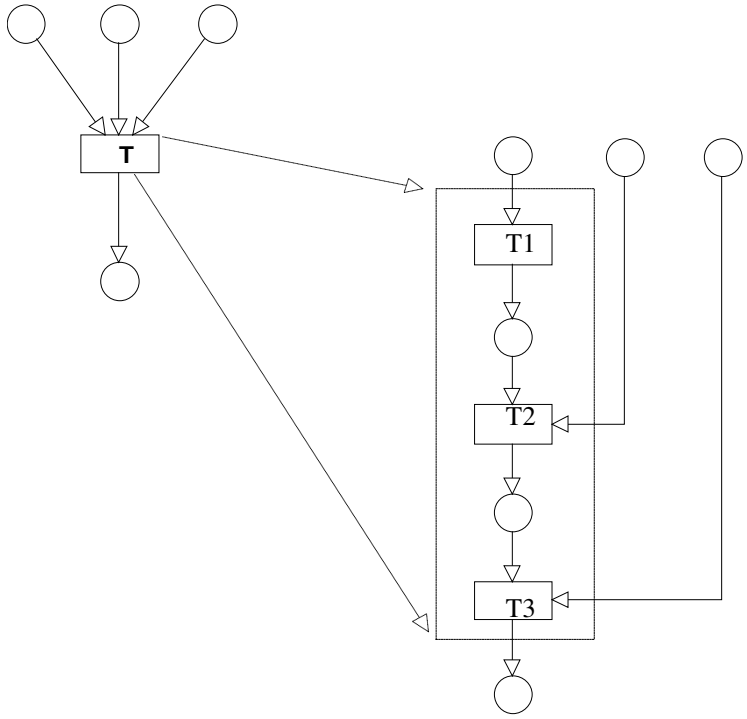


Abbildung 100: Transitionsverfeinerung

In der Abbildung 101 ist die Verfeinerung der List.forEach-Methode aus dem Listen-Beispiel von Wienberg dargestellt. Diese Methode überprüft, ob die gegebene Liste leer ist. Ist dies der Fall so wird die Methode beendet. Ist die Liste nicht leer, so wird auf jedes Element der Liste die map-Methode angewendet. Das Wesentliche in diesem Netz ist das fork-join-Konstrukt. Der Kontrollfluß wird durch die *fork*-Transition in zwei nebenläufige Ausführungspfade aufgespalten und nachdem die Transitionen *head* und *tail* und die jeweils nachfolgenden Transitionen geschaltet haben, durch die *join*-Transition wieder zusammengeführt. Auf der Oberseite wird diese Realisierung versteckt.

In dem Listen-Beispiel werden einige abkürzende Schreibweisen verwendet, um das Netz möglichst übersichtlich zu gestalten und von technischen Details zu abstrahieren. Die Raute in der *Class*-Stelle aus Abbildung 101 bezeichnet eine Stelle, die eine Instanzvariable enthält. Die Ellipsen in den Transitionen sind Nebenstellen der Transition. Die vermeintlich aus dem ‘Nichts’ kommenden Kanten sind mit der Transition ‘HS’ verbunden, die dafür sorgt, die Parameter des Methodenaufrufs als lokale Variablen zur Verfügung zu stellen.

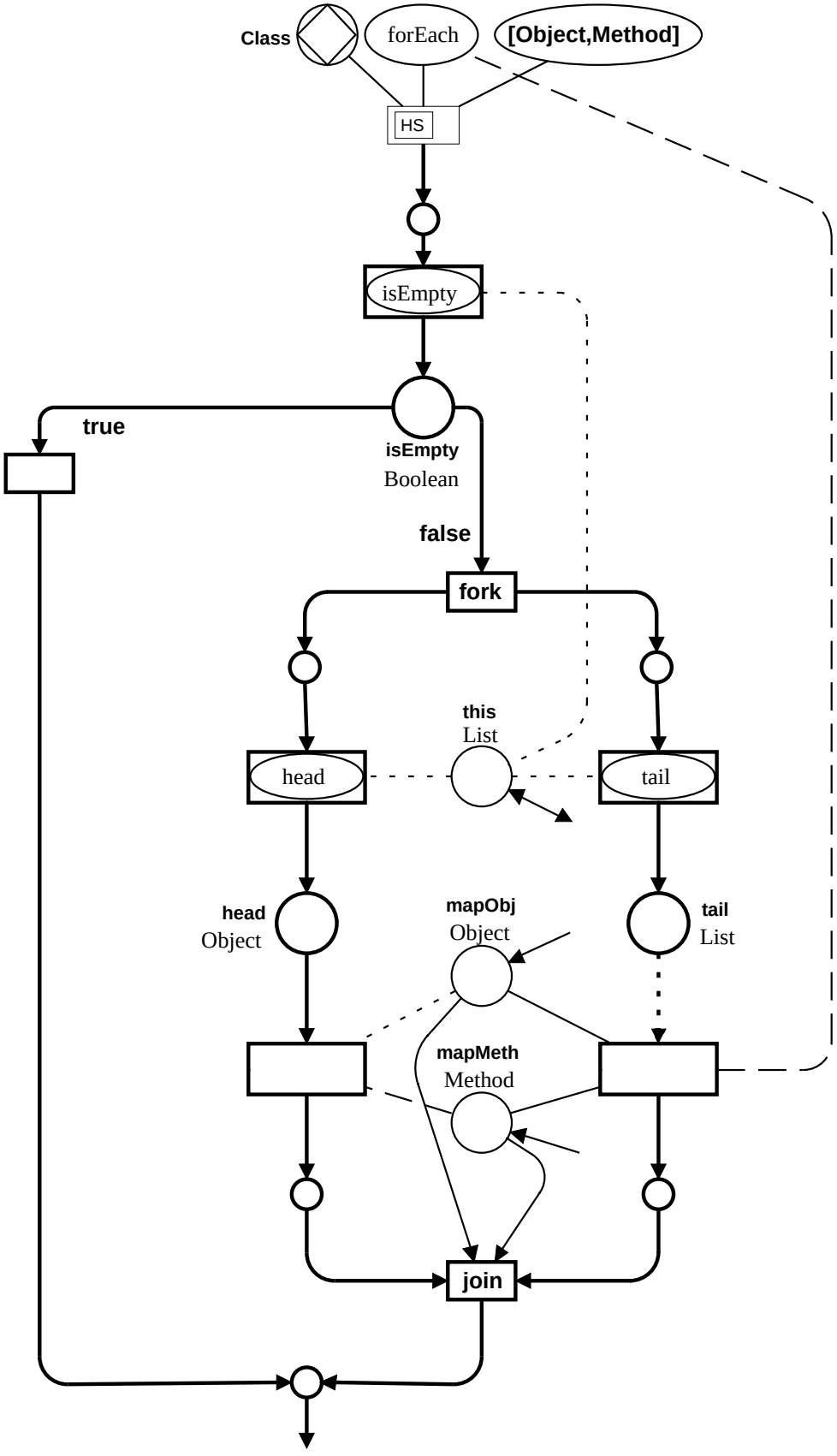


Abbildung 101: Verfeinerte Darstellung der List.forEach-Methode

4.4 Kommunikation

Zur Realisierung der Kommunikation zwischen Objekten gibt es verschiedene Ansätze, die in diesem Abschnitt vorgestellt werden. Möglich ist sowohl die Kommunikation in einem lokalen System, als auch die Kommunikation von Objekten in verteilten Umgebungen. Anhand eines Beispiels wird die Kommunikation zwischen zwei Objekten betrachtet und ausführlich auf den Methodenaufruf eingegangen.

4.4.1 Asynchrone Kommunikation

Die Kommunikation zwischen Objekten wird bei [Moldt 96] durch einen Nachrichtenhandler realisiert. Er ist dafür verantwortlich, die Nachrichten den richtigen Objekten zuzuweisen. Er kann beliebig komplex konzipiert werden, sodaß auch eine Kommunikation von Objekten in verteilten, nebenläufigen Umgebungen möglich ist. Nachrichten werden aus der out_pool Stelle eines sendenden Objekts genommen und in die in_pool Stelle des empfangenden Objekts gelegt. Für die lokale Lösung wird der Nachrichtenhandler dadurch realisiert, daß alle in_pool und out_pool Stellen zu einer Stelle verschmolzen werden. Diese Stelle wird als 'Global Fusion Place' mit der verwendeten Software 'Design/CPN' realisiert. Die Objekte 'lauschen' an dieser Stelle und filtern die für sie bestimmten Nachrichten heraus und legen Nachrichten an andere Objekte hinein. Die Abbildung 102 skizziert dieses Konzept.

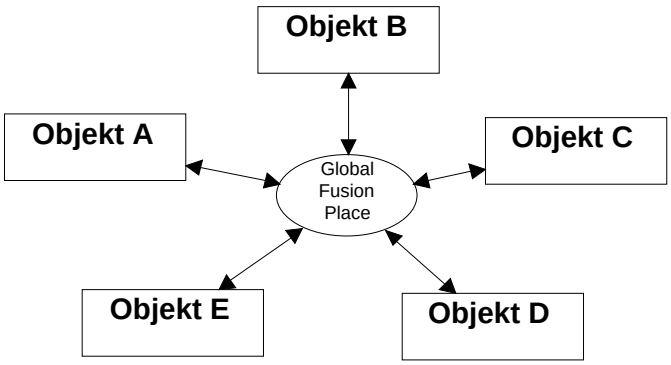


Abbildung 102: Nachrichtenaustausch über eine globale Fusions-Stelle

4.4.2 Simulation synchroner Kommunikation durch asynchrone Kommunikation

In [Maier 97] wird die Kommunikation zwischen zwei Objekten, 'Kunde' und 'Konto', wie in der folgenden Abbildung realisiert. Dargestellt ist der Aufruf der Konto-Methode 'buchen' seitens des Kunden-Objekts. Von der invoke-Transition (INV) im Kunden-Objekt wird eine Nachricht an die input-Transition (IN) im Bank-Objekt gesendet. Der einzuzahlende Betrag wird von dieser an die output-Transition (OUT) weitergereicht, die den Kontostand aktualisiert und diesen als Nachricht an den Kunden zurückschickt. Die receive-Transition (REC) nimmt die Antwortnachricht entgegen. Die Transitionen INV und REC können als vergröberte Transition INVREC bei der Modellierung verwendet werden. Der Name der rufenden Transition ist der Syntax der objektorientierten Programmierung angepaßt und wird hier als abkürzende Schreibweise verwendet und bedeutet, daß bei dem Objekt Konto mit der OID 'KtoId' die Methode 'buchen(x)' aufgerufen wird. Weitere abkürzende Schreibweisen sind in [Maier 97] nachzulesen. Die Kantenbeschriftungen haben nur nur lokale Gültigkeit, sodaß das 'y' beim Kunden nicht dem 'y' beim Konto entspricht. Durch die Marke in der Stelle 'wait' wird der Methodenaufruf synchronisiert.

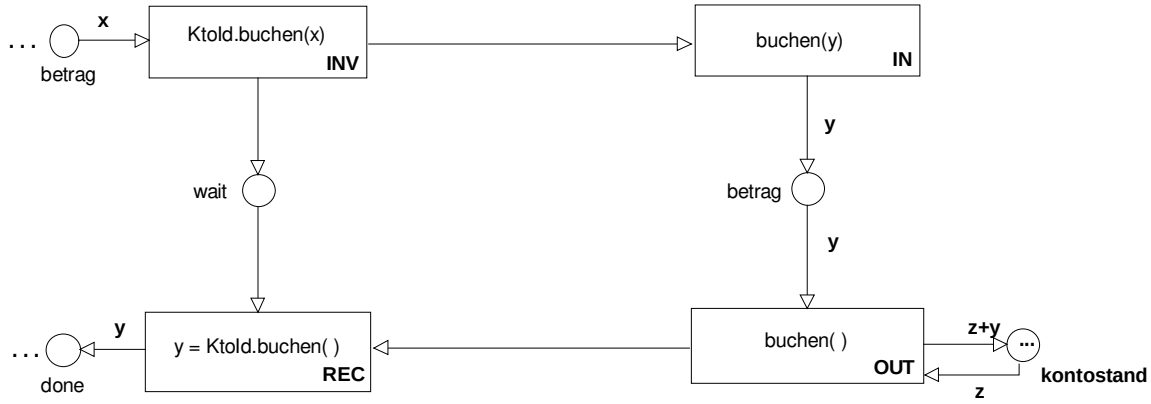


Abbildung 103: Kommunikation zwischen Objektnetzen nach [Maier 97]

4.4.3 Synchrone Kommunikation

In [KMW 98] werden alle in-pool-Stellen zu einer großen Stelle (Global Fusion Place) verschmolzen, die als 'CallMsgPool' bezeichnet wird. Die Verschmelzung aller out-pool-Stellen heißt 'ReturnMsgPool'. Die Abbildung 104 zeigt zunächst die vollständige Call-Seite, auf der die Nachrichten zusammengesetzt und in die Stelle *CallMsgPool* gelegt werden. Dieser Mechanismus wurde bereits im Abschnitt über das Nachrichtenformat beschrieben. Beim Versenden einer Nachricht wird eine eindeutige Aufrufidentifikation (callid) mittels einer globalen Variablen erzeugt. Die Nachrichten-ID wird in der Stelle 'Wait' gespeichert, um die Antwortnachricht eindeutig zu der abgeschickten Nachricht zuzuordnen. Durch diesen Mechanismus sind die schon beim Nachrichtenformat erwähnten nebenläufigen Methodenaufrufe möglich.

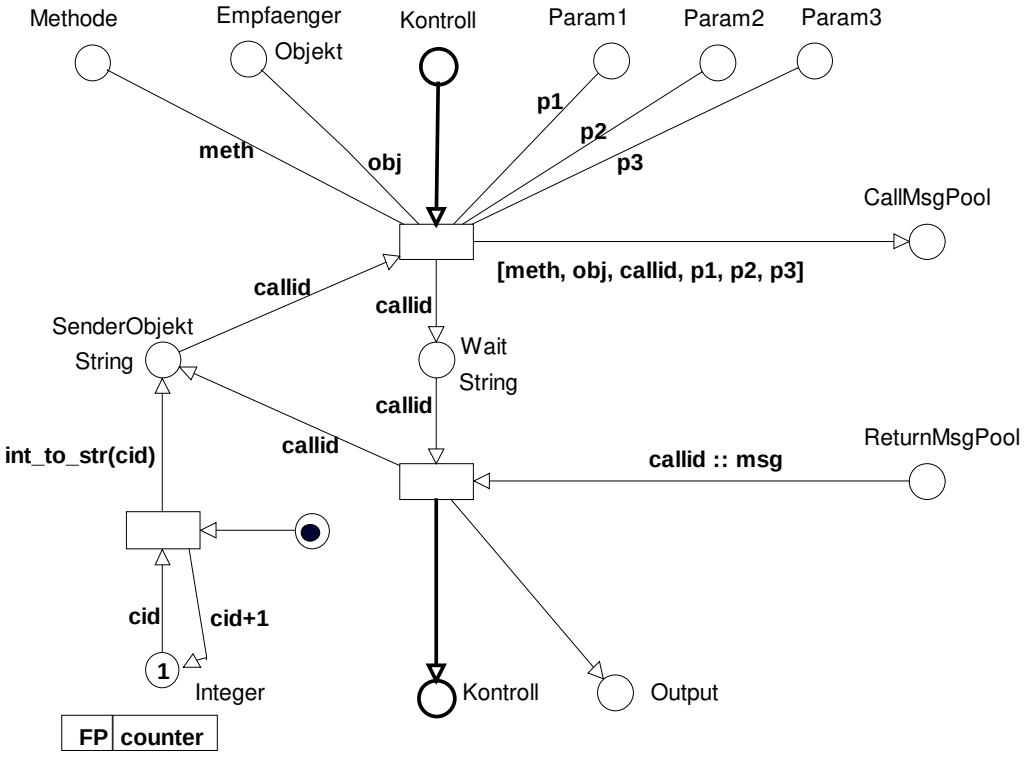


Abbildung 104: Call-Seite

4.4.4 Beispiel für synchrone Kommunikation und einen Methodenaufruf

Die Abbildung 105 gibt eine Vorstellung davon, wie der Aufruf einer Methode von einem Objekt aus einem anderen Objekt in [KMW 98] erfolgt. Dargestellt sind die folgenden vier Seiten: die Kunden-Seite, die Bank-Seite, die Call-Seite und die Methoden-Seite. Die gestrichelten Linien zwischen den Stellen bedeuten, daß die miteinander verbundenen Stellen identisch sind. Die Methode 'Einzahlen' aus dem Objekt 'Kunde' ruft bei dem Bank-Objekt die Methode 'Kontostand aktualisieren' auf. Dieser Methodenaufruf erfolgt über eine Nachricht, die auf der Call-Seite erzeugt wird. Diese Nachricht setzt sich aus den im Abschnitt über das Nachrichtenformat vorgestellten Komponenten zusammen. Die Nachricht wird von der Methoden-Seite empfangen und in ihre Bestandteile aufgeteilt. Diese werden in die Eingangsstellen der Methode 'Konto aktualisieren' gelegt, um diese Methode auszuführen. Nach der Methodenausführung wird auf der Methoden-Seite die Antwortnachricht zusammengebaut, die von der Call-Seite empfangen und an das Objekt 'Kunde' weitergeleitet wird.

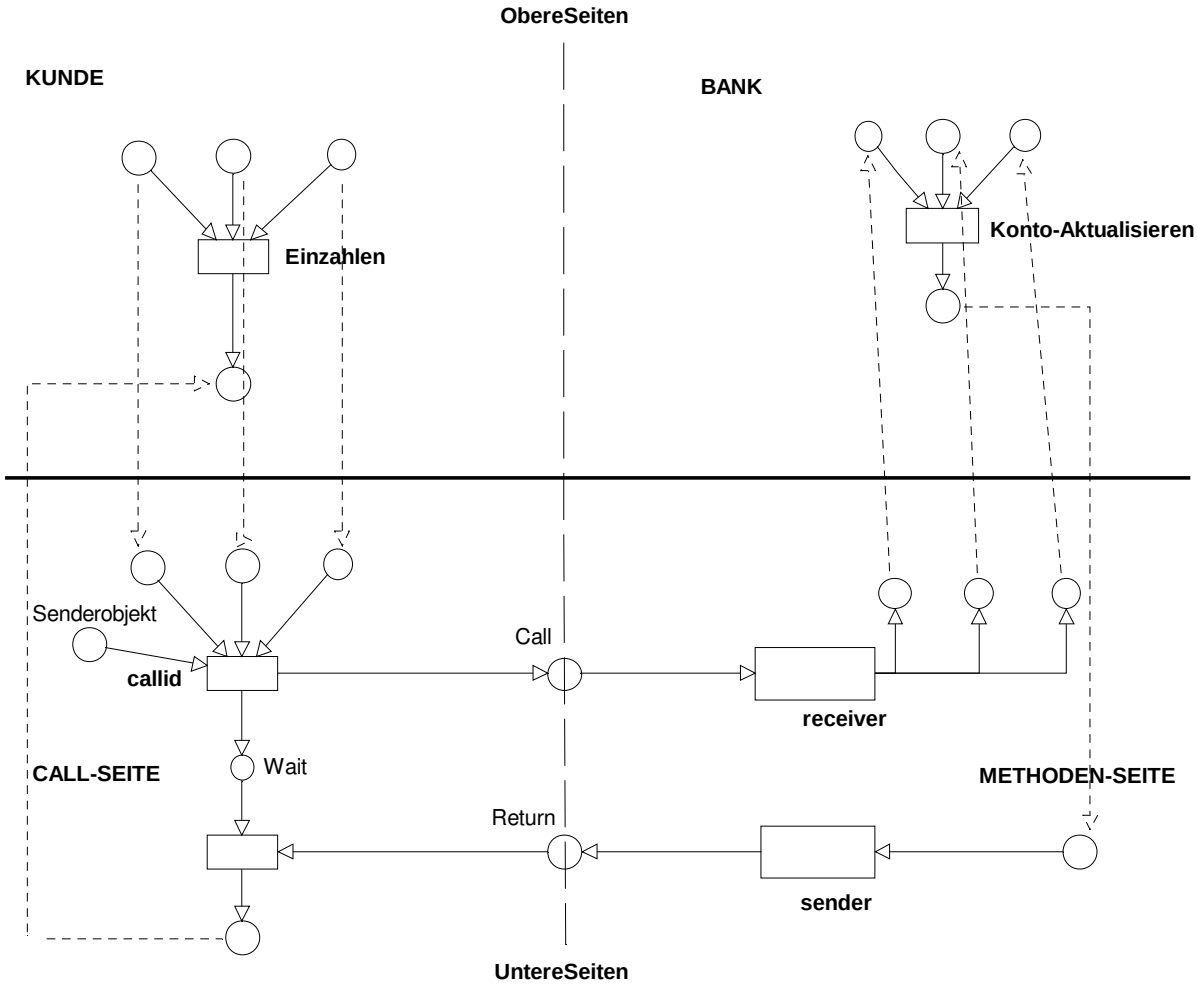


Abbildung 105: Aufrufstruktur

4.5 Assoziationen, Multiplizität und Aggregation

In [Rumbaugh et al. 91] werden Assoziation als Beziehungen zwischen Objekten bzw. Objektklassen definiert. Die verschiedenen Arten von Assoziationen werden in diesem Abschnitt nochmals kurz dargestellt. Nach der Betrachtung der Konzepte werden Petrinetzrealisierungen vorgestellt.

4.5.1 Assoziationsklasse

Als Assoziationsklasse werden Beziehungen zwischen Objekten insbesondere dann dargestellt, wenn sie eigene Attribute und Methoden haben. Ein oft in der Literatur verwendetes Beispiel ist das Arbeitsverhältnis zwischen einer Firma und einer Person. Das Attribut ‘Gehalt’ wird als Attribut der Assoziationsklasse modelliert und nicht als Attribut der Person. Der Zweck ist eine saubere Modellierung und Erhöhung der Wiederverwendbarkeit. Abbildung 106 zeigt die Notation einer Assoziationsklasse in der Notation der Object Modeling Technique (OMT). Eine Firma kann keine, eine oder viele Personen beschäftigen und eine Person ist bei keiner oder einer Firma angestellt. Zu einem Arbeitsverhältnis gehört das Attribut ‘Gehalt’ und die Methode ‘ändere_Gehalt’.

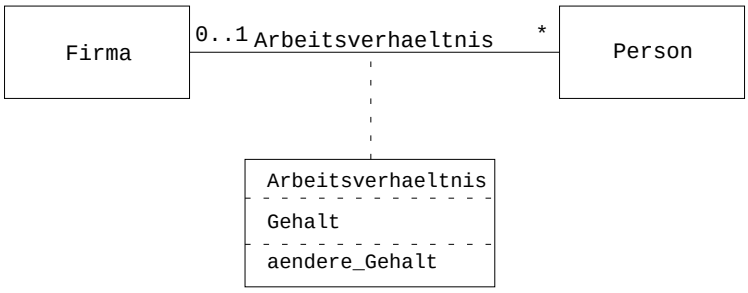


Abbildung 106: Assoziationsklasse in OMT-Notation

Die Netzdarstellung in der Abbildung 107 zeigt eine Assoziationsklasse aus [Moldt 96]. Die Beziehungen zwischen einer Person und einer Firma wird als Marken auf Stellen modelliert. Die Tupel in den Stellen ‘Firma’ und ‘Person’ sind durch den ersten Tupelteil einander zugeordnet. Zum Beispiel arbeitet Person ‘12’ für Firma ‘6’. Dies ist das Arbeitsverhältnis ‘(AV, 1) ’. In der Stelle ‘Gehalt’ ist das zu einem Arbeitsverhältnis gehörende Gehalt festgelegt. Für unser Beispiel heißt dies, daß für das Arbeitsverhältnis ‘(AV, 1) ’ ein Gehalt von ‘4000’ festgelegt ist. Damit erhält Person ‘12’ bei Firma ‘6’ ein Gehalt von ‘4000’. Die Methode ‘ändere_Gehalt’ ist eine Methode der Assoziationsklasse. Die Methoden ‘Firma’ und ‘Person’ werden für die Klassen ‘Firma’ und ‘Person’ bereitgestellt, damit Objekte dieser Klassen Information über ihre Arbeitsverhältnisse bekommen können. Bei der Erstellung eines Arbeitsverhältnisses werden die entsprechenden Marken generiert und in den Stellen abgelegt. Wird in diesem Beispiel ein Arbeitsverhältnis aufgelöst, so werden die dieses Arbeitsverhältnis definierenden Marken in den Stellen gelöscht.

In Abhängigkeit von der Art der Assoziationen werden unterschiedliche Protokolle implementiert. Auf diese wird im folgenden Unterkapitel eingegangen, wenn der Unterschied zwischen Kann- und Muß-Beziehungen erläutert wird.

Bei einfachen Assoziationen, die nicht als Assoziationsklasse modelliert werden, genügt eine Stelle in dem einen Objektnetz, in der die Objekt-ID des zu diesem Objekt in einer Beziehung stehenden anderen Objekts abgelegt ist. Dadurch kennt dann das eine Objekt das andere.

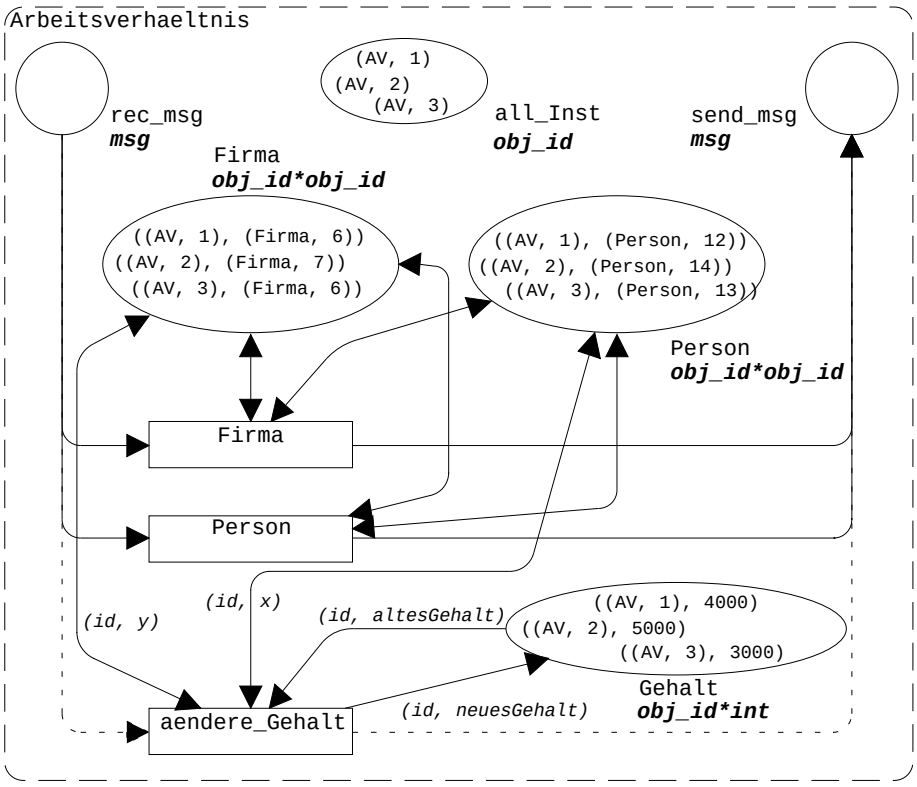


Abbildung 107: Assoziationsklasse als Petrinetz

4.5.2 Assoziationen und Multiplizität

Die an den Assoziationsenden angegebenen Multiplizitäten geben an, wieviele Objekte, die in dieser Beziehung stehen, existieren können bzw. müssen. Kann-Beziehungen erkennt man an dem Einschluß der ‘0’ im Multiplizitätsbereich. Ist diese ausgeschlossen, so handelt es sich um eine Muß-Beziehung. Eine Übersicht findet man in [Rumbaugh et al. 91]. In der obigen Beispiel-Assoziation kann eine Person in einer oder keiner Firma arbeiten und eine Firma stellt keine oder beliebig viele Personen an.

Ein weiteres Beispiel in Abbildung 108 zeigt eine Muß-Beziehung aus der Sicht der Person, die besagt, daß es zu einer Person genau eine Adresse geben muß. Dieser Umstand folgt aus dem unbeschrifteten Assoziationsende auf der Seite der Person. Aus der Sicht der Adresse ist es eine Kann-Beziehung, denn zu einer Adresse kann es keine, eine oder viele Personen geben. Das Zeichen ‘*’ am Assoziationsende auf der Seite der Adresse steht stellvertretend für diese Alternativen.

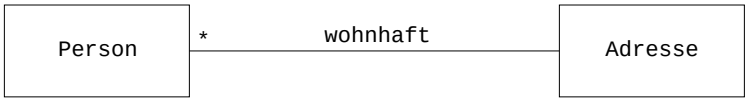


Abbildung 108: Kann- und Muß-Assoziation

Die Kann- und Muß-Beziehungen werden in den Netzen entsprechend abgebildet. Bei einer Muß-Beziehung wird bei der Erzeugung eines Objekts in der New-Methode die benötigte Beziehung initiiert. Wird ein Objekt gelöscht, das in einer solchen Beziehung zu einem anderen Objekt steht, so muß dafür gesorgt werden, daß die Beziehung auch gelöscht wird. Beispielhaft wird das Einstellen

einer Person in einer Firma in [Moldt 96] durchgeführt. Da es dort zu der Person eine Adresse geben muß, wird in der new-Methode der Klasse ‘Person’ die Assoziation zu der Adresse eingerichtet.

Die Multiplizitäten ‘0..n’, ‘1..n’ und ‘n..m’ definieren weitere Kann- bzw. Muß-Beziehungen. In der Netzdarstellung wird eine ‘0..n’ dadurch modelliert, daß bei dem Aufruf eines Konstruktors eines in einer solchen Beziehung stehenden Objekts keine Assoziation erzeugt werden muß.

Die o.g. Assoziation zwischen Person und Adresse wird als einfache Assoziation bezeichnet, da sie nicht als Assoziationsklasse modelliert wird. In der Netzdarstellung ist in dem Person-Objektnetz eine Stelle enthalten, in der die Objekt-ID der zu einer Person gehörenden Adresse abgelegt ist. Damit hat ein Objekt vom Typ ‘Person’ Zugriff auf seine Adresse.

4.5.3 Aggregation

Die Aggregation ist eine spezielle Form der Assoziation und bezeichnet eine ‘besteht aus’ oder ‘ist Teil von’ Beziehung. Diese Relation ist transitiv und antisymmetrisch. Ein Aggregat kann zusätzliche Funktionen erbringen, die von den einzelnen Teilen nicht erbracht werden. Keine Angabe der Multiplizität bedeutet, daß es sich um die Multiplizität ‘1’ handelt. In [Rumbaugh et al. 91] wird die Aggregation wie in Abbildung 6 in Kapitel 2.1 dargestellt.

Die Darstellung der aggregierten Objekte als Objektnetze variiert gegenüber der Darstellung der anderen Assoziationen. Die Transition `input` erhält eine Stelle als Nebenbedingung, die die Objekt-IDs der Objekte enthält, die auf dieses aggregierte Objekt zugreifen können. Möchte man diese Zugriffsberechtigungen dynamisch verändern, kann noch eine spezielle Zugriffsberechtigung für das aggregierende Objekt eingerichtet werden, mittels derer es auf diese Nebenbedingung zugreifen kann. Die Darstellung der aggregierenden Objekte wird nicht verändert.

4.6 Vererbung und Polymorphismus

Vererbung ist die Basis für die Erhöhung der Wiederverwendbarkeit schon bestehender Klassen. Ermöglicht wird die gemeinsame Nutzung von Daten und Operationen durch verschiedene Klassen. Die Konzepte der Vererbung werden kurz aus programmiersprachlicher Sicht erläutert und dann auf Petrinetze übertragen.

4.6.1 Vererbungskonzepte

Klassen, die in einer Vererbungsbeziehung stehen, unterteilt man in Oberklassen und Unterklassen. Die Oberklasse enthält Attribute und Operationen, die die Unterklassen von der Oberklasse erben. Weitere Attribute und Operationen können den Unterklassen hinzugefügt werden. Die allgemeine Darstellungsweise nach [Rumbaugh et al. 91] zeigt die Abbildung 7 in Kapitel 2.1.

Verwendet wird in dieser Arbeit die spezielle Form der Vererbung nach dem Prinzip der Einsetzbarkeit (Subtyp-Prinzip). Demnach darf ein Objekt einer Unterklasse überall dort benutzt werden, wo ein Objekt der Oberklasse zulässig ist. Eine Einschränkung der Oberklasse in einer Unterklasse ist nicht zulässig.

4.6.2 Netzdarstellung

Dargestellt werden Ober- und Unterklassen, analog zur Klassenhierarchie, in [Moldt 96] als getrennte Petrinetze. Dabei wird die Schnittstelle der Unterklasse um die Schnittstelle der Oberklasse

erweitert. Um den Aufruf geerbter Methoden zu realisieren, wird in der Oberklasse eine weitere input-Transition eingerichtet, die Nachrichten der Unterklasse entgegennimmt. Objekte der Unterklasse können über die privaten Methoden der Oberklasse Zugriff auf die Attribute der Oberklasse haben. Bestehende Assoziationen der Oberklasse werden von den Unterklassen übernommen. Weitere Netzteile zur Verwaltung der Objekte der Unterklasse werden in den Oberklassen eingerichtet.

Durch die Vererbungshierarchie wird ein Objekt durch mehrere Objektnetze (Objektteile) dargestellt. Objekte dieser Art werden als ‘split objects’ bezeichnet. Der in dieser Hierarchie unten liegende Objektteil wird als Repräsentant des gesamten Objekts betrachtet. In dem folgenden Abschnitt ist in der Abbildung 109 eine zweistufige Hierarchie dargestellt.

4.6.3 Methodenaufruf

Als Beispiel für eine einfache Vererbung dienen hier die Klassen Figur und Linie aus Abbildung 109. Linie erbt von der Oberklasse Figur das Attribut Farbe und die Methoden ‘`zeichnen`’ und ‘`farbe`’. Die Unterklasse Linie erweitert die Oberklasse um die Methode ‘`länge`’. Ein Objekt der Klasse Linie wird durch zwei Objektnetze, wie in Abbildung 109, dargestellt.

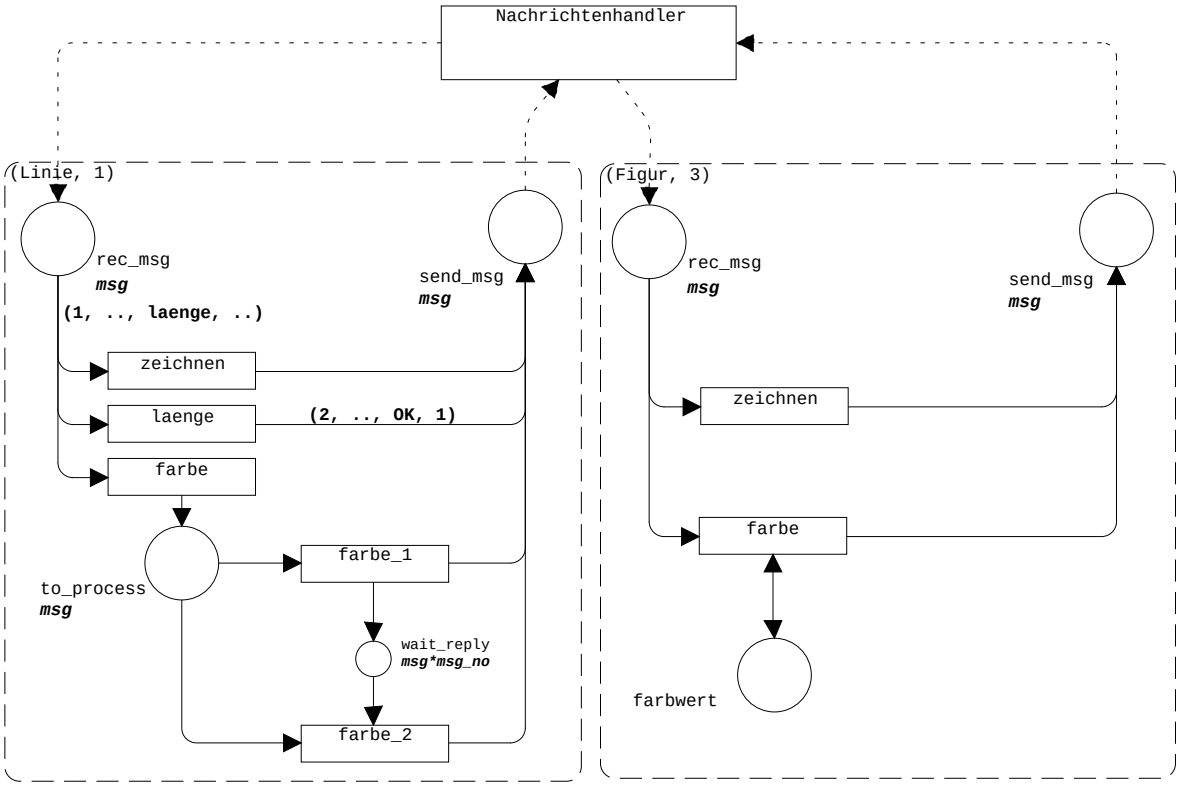


Abbildung 109: Vererbung in Netz-Notation

In [Moldt 96] werden verschiedene Methodenaufrufe dargestellt. Bei dem Aufruf einer neu implementierten Methode wird die Implementation im entsprechenden Objektteil ausgeführt. Der Oberklassen-Objektteil wird nicht benötigt. In der Abbildung 109 ist dies die Methode ‘`länge`’. Analog dazu wird ein Aufruf einer redefinierten Methode ohne Berücksichtigung des Oberklassen-Objektteils ausgeführt. Handelt es sich in der Oberklasse um eine abstrakte Methode, so muß diese in der Unterklasse implementiert werden, um aufgerufen werden zu können. Die Methode ‘`zeichnen`’ ist hier solch ein Fall.

Der Aufruf einer geerbten Methode erfolgt etwas anders. Die im Unterklassen-Objektteil aufgerufene Methode reicht die erhaltene Nachricht an den Oberklassenteil weiter. Dieser Vorgang wird als 'Delegation' bezeichnet. Im Oberklassenteil wird die Methode ausgeführt und das Ergebnis an den Unterklassenteil zurückgegeben. Analog ist das Vorgehen bei dem Aufruf von geerbten, unveränderten oder umbenannten Methoden. Zur Unterscheidung dieser Arten von Methodenaufrufen werden spezielle Eingangstransitionen bei den Klassennetzen verwendet. Einzelheiten hierzu findet man in [Moldt 96].

Ein anderes Konzept, das zur Modellierung von Vererbung eingesetzt werden kann, ist das der Interfaces. In der Abbildung 110 ist die Verwendung eines Interface dargestellt. Die Klasse I ist ein Interface, das von der Klasse B implementiert wird. Die Klasse B erbt von Klasse A, bei der es sich nicht um ein Interface handelt. Eine ausführliche Diskussion der Interface-Konzepte findet man in [Cornell 97].

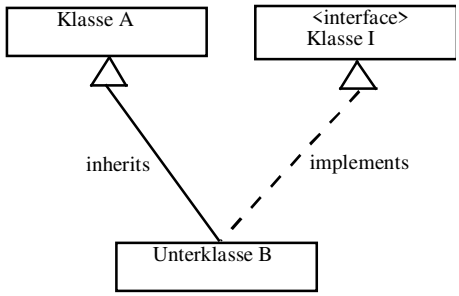


Abbildung 110: Vererbung mit einem Interface

4.6.4 Erzeugung von Objekten

Bei der Erzeugung eines Objekts, das in einer Vererbungshierarchie steht, müssen alle oben erwähnten Netzteile generiert werden. Das Verhalten und der Zustand eines Objekts wird durch die Vereinigung der Netzteile realisiert. Der Aufruf der new-Methode in der Unterklasse bewirkt die Ausführung dieser Methode dort und über ein Protokoll der new-Methode in der Oberklasse. Ein Objekt wird entsprechend der Vererbungshierarchie durch mehrere Objekt-IDs repräsentiert. In dem Objekt der Oberklasse werden die Objekt-IDs der erzeugten Objekte der Unterklasse zusammen mit der eigenen ID gespeichert. Ein Objekt wird also in der Oberklasse durch ein Tupel bestehend aus der ID der Unterklasse, evtl. einer weiteren Oberklasse, der eigenen ID und dem 'Einstiegsobjekt' gespeichert. Diese ID-Tupel sind für jede Ebene in der Hierarchie notwendig. Details und ein Beispiel findet man in [Moldt 96].

Bei der Verwendung des Interface-Konzeptes müssen neben der Erzeugung des Interface sichergestellt sein, daß alle Methoden des Interface implementiert werden. Einzelheiten zur Erzeugung von Objekten unter Verwendung des Interface-Konzeptes wurden bereits zu Beginn dieses Kapitels erläutert.

4.6.5 Vererbung von Assoziationen

Assoziationen werden mitvererbt, d.h. eine Klasse erbt alle Assoziationen von der vererbenden Klasse. Als Beispiel dient hier die Abbildung 111 aus [Moldt 96]. Die Assoziation zwischen Objekten der Klassen 'Firma' und 'Person' wird von Objekten der erbenden Klassen, hier 'Universität' und 'Wissenschaftlicher_Mitarbeiter', mitgeerbt. Die Modellierung von Assoziationen wurde bereits beschrie-

ben. Eine explizite Netzdarstellung wird hier nicht aufgezeigt. Sie erfolgt analog zu den in diesen Kapiteln dargestellten Umsetzungen der Konzepte zur Vererbung und Assoziation. Die dort unterschiedlichen Protokolle müssen implementiert werden. Diese können, je nach Art der Assoziation, sehr komplex sein. Eine detaillierte Betrachtung geht über die Zielsetzung dieser Arbeit hinaus. Behandelt wird dieses Thema ausführlich in [Rölke 99].

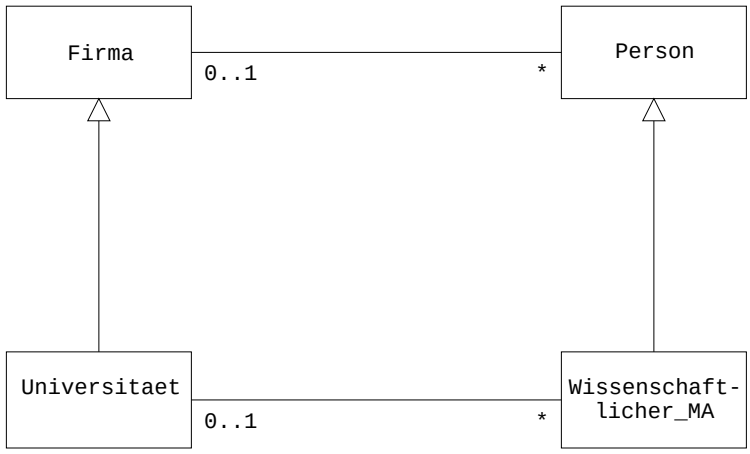


Abbildung 111: Vererbung einer Assoziation

4.6.6 Polymorphismus

Als ‘polymorph’ bezeichnet man Methoden, wenn diese auf beliebige Typen angewendet werden können. Als ‘überladen’ werden Methoden bezeichnet, die für verschiede, aber nicht beliebig viele, Typen definiert sind. Durch die Vererbung und dynamische Bindung bei der Objektorientierung wird implizit Polymorphismus bereitgestellt.

Dies gilt inbesondere für den sogenannten ad-hoc-Polymorphismus (Überladen), weil das Aufrufen einer Methode eines Objekts automatisch diejenige Methode wählt, die sich auf die Klasse des aktuellen Objekts stützt. Parametrischer Polymorphismus wird in objektorientierten Sprachen durch spezielle Mechanismen zur Parametrisierung von Klassen realisiert. Die Methoden müssen so implementiert werden, daß sie Referenzen auf beliebige Objekte als Parameter erhalten.

Mit den objektorientierten Petrinetzen können beide Arten von Polymorphismus umgesetzt werden. Beim Überladen werden gleichbenannte Methoden in unterschiedlichen Klassen einfach unterschiedlich auf verschiedenen Methoden-Seiten implementiert. Die Semantik sollte wegen der Verständlichkeit bei der Verwendung gleicher Namen erhalten bleiben. Parametrischer Polymorphismus kann dadurch erreicht werden, daß die Methoden einer Klasse Referenzen auf beliebige Objekte als Parameter erhalten.

4.7 Beispiel: Listenklassen

Anhand eines Beispiels werden einige der oben vorgestellten Konzepte veranschaulicht. Dieses Beispiel zur Listenerzeugung und -verwaltung wird zunächst textuell beschrieben. Nach der Betrachtung des dazugehörenden Klassendiagramms werden einige Netz-Seiten ausführlich diskutiert.

4.7.1 Beschreibung des Listenbeispiels

Der Typ Liste (das Interface ‘List’) wird durch zwei Klassen implementiert, die Klasse ‘Elist’ und die Klasse ‘NEList’. Der Name ‘EList’ steht für die leere Liste und der Name ‘NEList’ für die nichtleere Liste. Die definierten Methoden der Klassen verändern nicht die bestehenden Listen, sondern instanziieren neue Listen. Dieses Konzept entspricht der Implementation von Listen in der funktionalen Programmierung. In der Abbildung 112 werden die genannten Listen in der Box-und-Pointer Notation dargestellt.

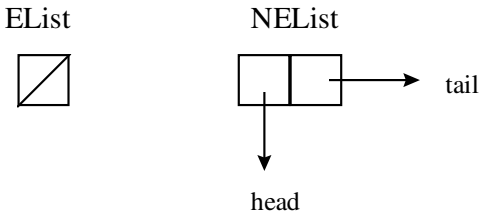


Abbildung 112: Box-Pointer Notation für Listen

Als Beispiel für die Verwendung von Methoden wird hier die add-Methode betrachtet. Der Aufruf dieser Methode liefert eine nichtleere Liste (NEList) zurück. Bei dieser handelt es sich um eine neue Liste mit einer neuen ID. Die ursprüngliche Liste bleibt unverändert. Dargestellt sind beide Listen in der folgenden Abbildung. Aufgerufen wird die add-Methode mit dem Parameter ‘c’ der Liste ‘L1’. Das Resultat ist die Liste ‘L2’.

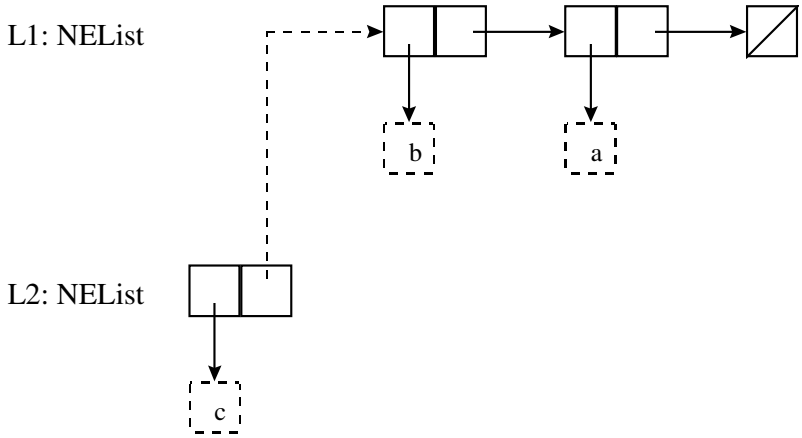


Abbildung 113: Aufruf einer add-Methode

Die Abbildung 114 stellt die im Beispiel verwendeten Klassen dar. Bei den Klassen ‘Pair’, ‘List’, ‘Constructor’ und ‘Tester’ handelt es sich um Interfaces. Implementiert werden sie von den Klassen ‘NEList’, ‘Elist’ und ‘Test’. Die Klassen ‘NEList’ und ‘EList’ implementieren beide das Interface ‘List’. Zu einer nichtleeren Liste gibt es immer genau eine leere Liste, während es zu einer leeren Liste keine oder viele nichtleeren Listen geben kann. Zu Testzwecken wird eine Klasse ‘Test’ implementiert. Durch die main-Methode eines TestObjekts werden einige Methoden der Listen aufgerufen. Dies sind die Methoden ‘new’, ‘add’, ‘filter’ und ‘show’.

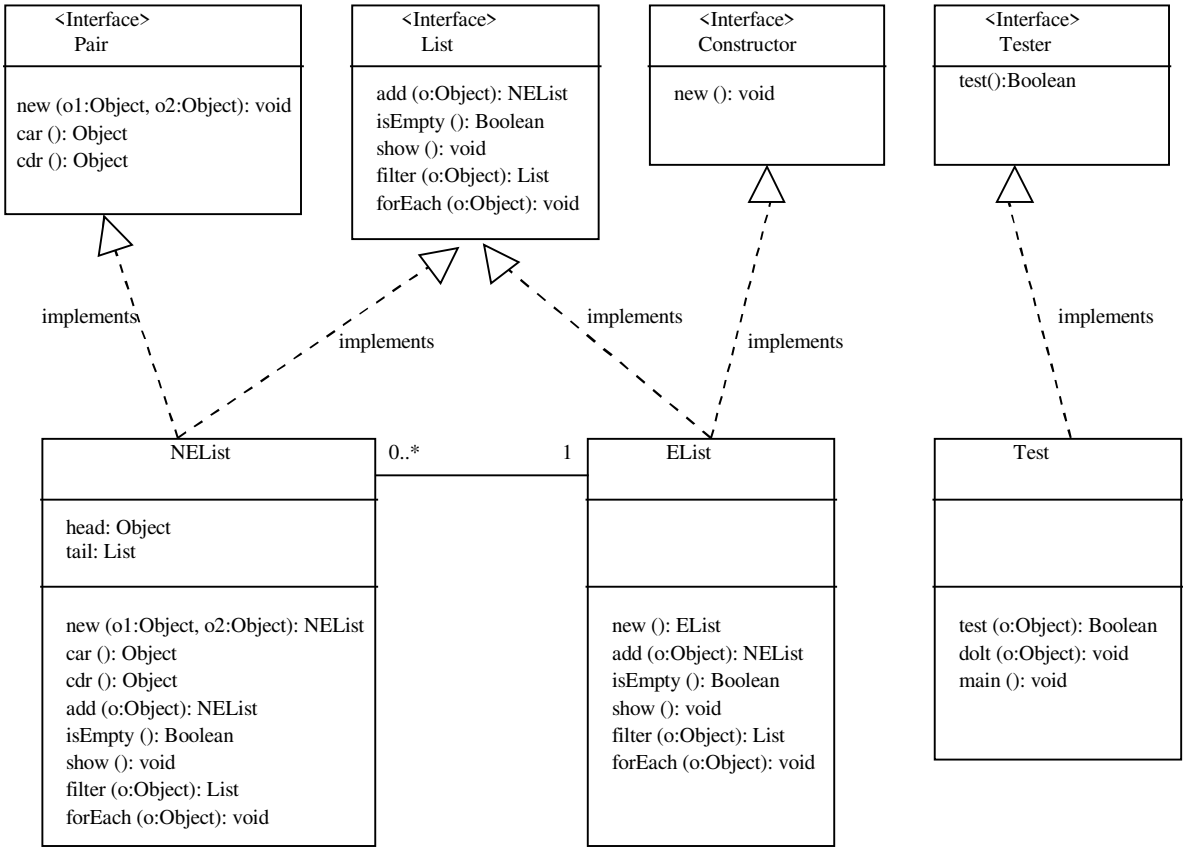


Abbildung 114: Klassendiagramm

Die in den Interfaces genannten new-Methoden dienen zur Initialisierung erzeugter Objekte. Tatsächlich aufgerufen wird die Klassenmethode ‘new’, die ein Objekt der entsprechenden Klasse erzeugt und dann dessen new-Methode aufruft.

4.7.2 Netzhierarchie

Die Hierarchieseite in Abbildung 115 zeigt zunächst alle verwendeten Netz-Seiten. Die hierarchischen Beziehungen zwischen diesen Seiten sind durch gerichtete Kanten dargestellt. Die Seiten #2 (EList) und #15 (NEList) entsprechen den Listenklassen. Sie verwenden als Unterseiten die Interfaces (Seiten #26, #27, #29), die Class-Seite (Seite #5) und die Methoden-Seiten (#3, #13, #18, #19, #21). Die Methoden-Seiten benutzen die Unterseite Call (Seite #20). Das Interface Tester und die dieses Interface implementierende Klasse mit der main-Methode werden nicht explizit diskutiert, weil diese Seiten nur zum Testen der Listenmethoden eingeführt wurden. Erwähnenswert ist, daß alle Klassen-Netze die Class-Seite als Unterseite verwenden und alle Methoden-Seiten, die andere Methoden aufrufen, die Call-Seite.

Weiterhin ist hier ein Beispiel von Code-Wiederverwendung zu sehen: NEList und Elist benutzen dieselbe Implementation der add- (Seite #13) und der show- (Seite #21) Methode.

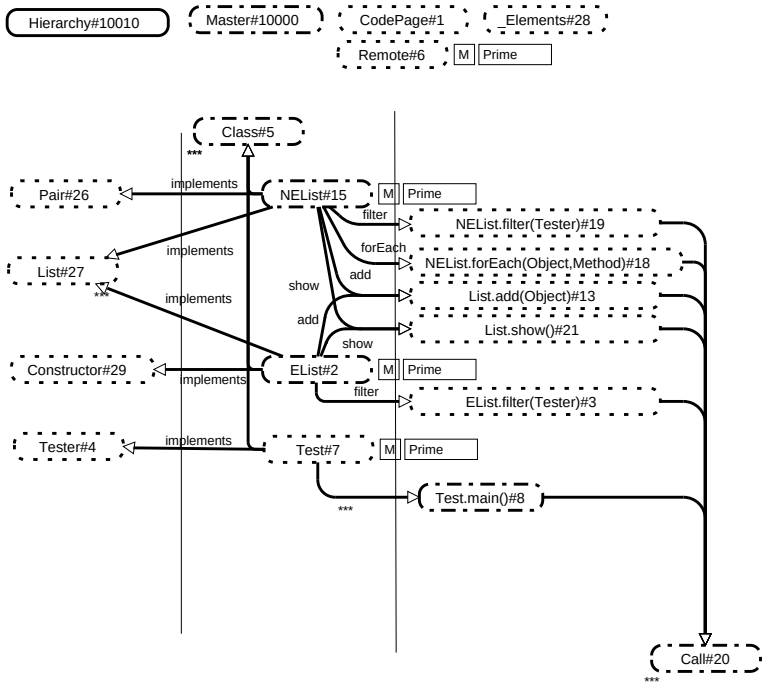


Abbildung 115: Hierarchie-Seite

4.7.3 Interfaces

Die Interfaces definieren die zu implementierenden Methoden. Festgelegt wird der Methodenname, die Parameter mit Typangabe und der Rückgabetyp der Methoden. Die Interfaces des Listenbeispiels werden hier vorgestellt. Ausführlich diskutiert wird an dieser Stelle nur das List-Interface.

Das List-Interface definiert die Methoden:

- add(o:Object):NEList
- isEmpty():Boolean
- show():void
- filter(o:Object):List
- forEach(o:Object):void

Für jede Methode gibt es eine Transition, die aus den Nachrichten in der Stelle inMsg diejenigen erkennt, die für die Methode bestimmt sind. Exemplarisch wird hier die add-Methode in Abbildung 116 betrachtet. Neben dem Filtern von Nachrichten ist es die Aufgabe der o.g. Transition, die Parameter aus der Nachricht in dafür vorgesehene Stellen zu legen. Im Falle der add-Methode ist dies newHead in der Nachricht ‘(call,[I(class,oid),M”add”,newHead])’. Mit ‘call’ wird der Sender bezeichnet, ‘I(class,oid)’ den Empfänger und ‘M”add”’ die aufgerufene Methode. Die Stelle ‘add_c’ repräsentiert den Kontrollfluß. Eine Marke in dieser Stelle zeigt an, daß die Methode gerade aufgerufen wird. In der Stelle add_this wird das Objekt, dessen add-Methode aufgerufen wurde, abgelegt. Der Rückgabewert der Methode ist ein nichtleeres Listenobjekt. Dieses ist nach der Abarbeitung der Methode in der Stelle ‘add’ enthalten. Die Transition auf der rechten Seite erzeugt die Antwortnachricht unter Verwendung des Ergebnisses. Die Stellen, auf die die Parameter zu Beginn gelegt wurden, sowie die Kontrollflußstelle und die Stelle mit der Referenz auf das Objekt, werden von dieser Transition geleert. Die Antwortnachricht wird auf die Stelle ‘ReturnMsg’ gelegt.

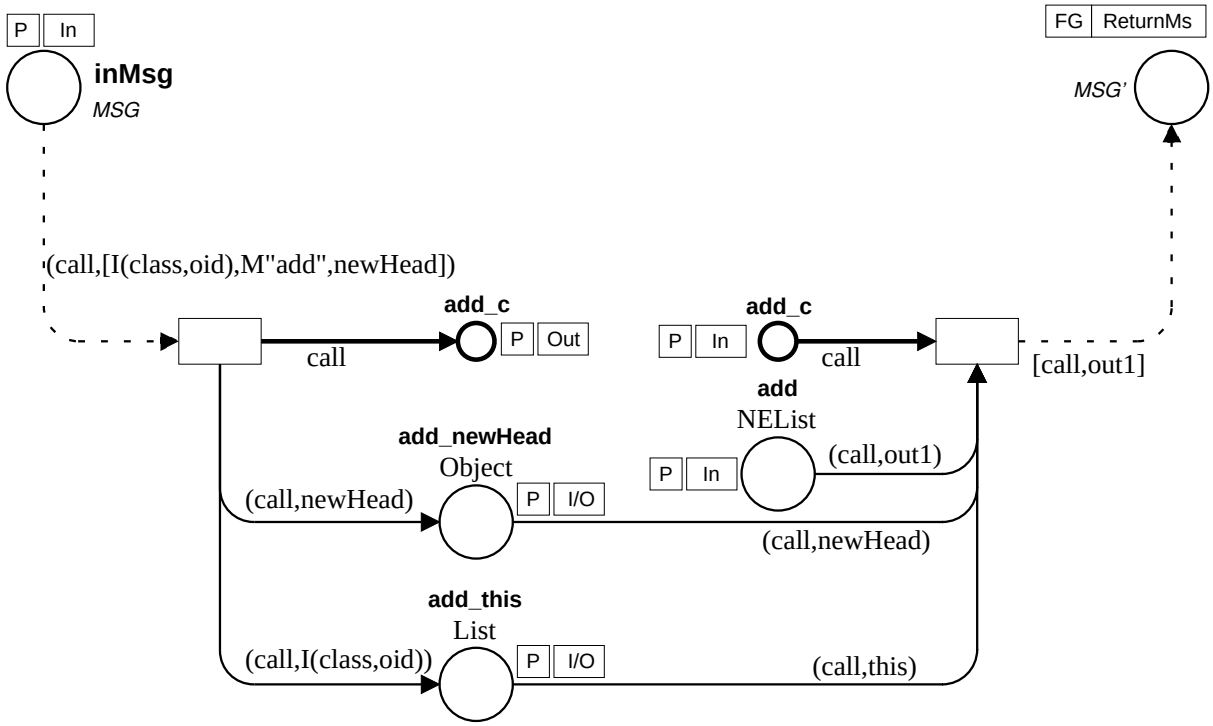


Abbildung 116: Schnittstelle der add-Methode

Das Pair-Interface definiert die Methoden:

- new(o1:Object,o2:Object):void
- car():Object
- cdr():Object

Das Constructor-Interface definiert die Methode:

new():void

4.7.4 Listenklassen

Die Klasse ‘EList’ (Netz-Seite #2) implementiert die Interfaces ‘List’ (Netz-Seite #27) und ‘Constructor’ (Netz-Seite #29). D.h. sie implementiert die Methoden ‘new()’, ‘add(o:Object)’, ‘isEmpty()’, ‘show()’, ‘filter(o:Object)’ und ‘forEach(o:Object)’. Dargestellt ist der Ausschnitt der Klasse ‘EList’ mit der new- und der add-Methode in der Abbildung 117.

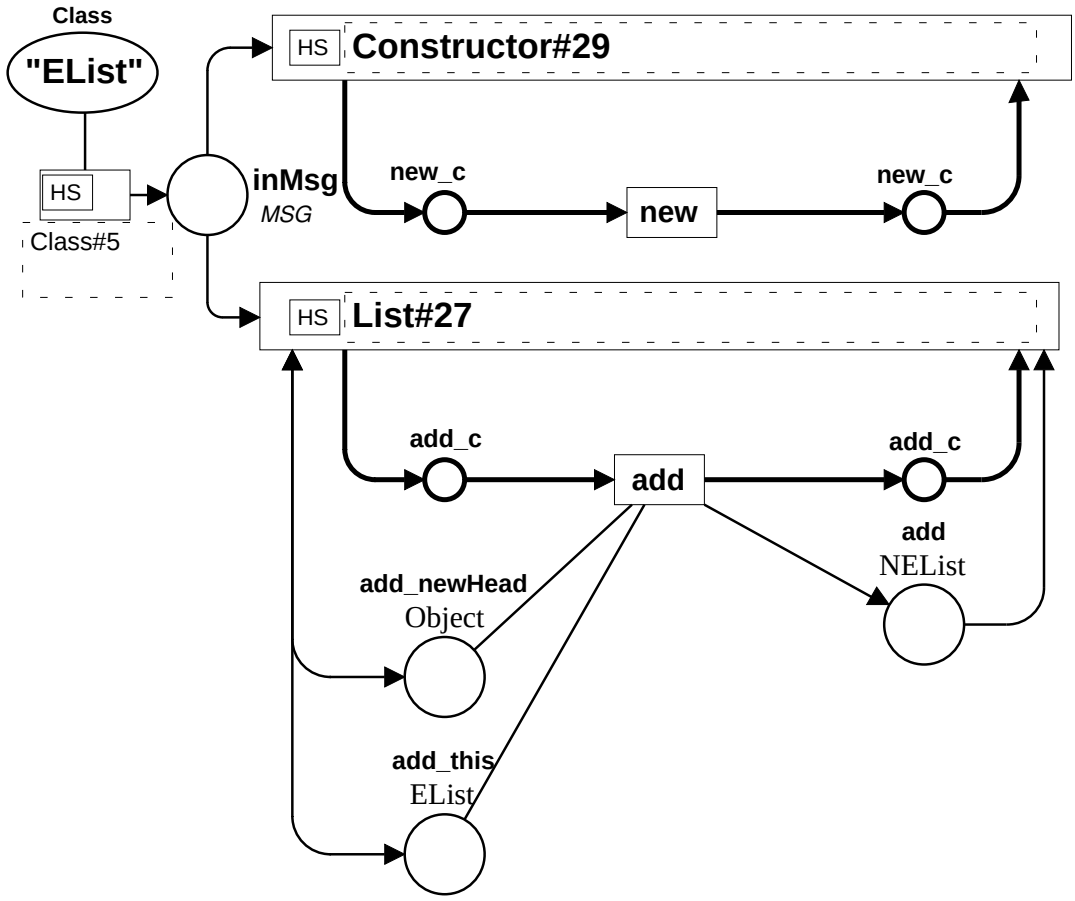


Abbildung 117: Ausschnitt der Implementationsklasse EList

Die Klasse ‘EList’ nimmt nur Nachrichten entgegen, die für sie bestimmt sind. Diese Selektion erfolgt auf der Unterseite ‘Class’, die in der Abbildung 117 durch die Transition auf der linken Seite repräsentiert wird. Die Class-Seite unterscheidet auch zwischen Nachrichten an die Klasse und Nachrichten an Objekte. Die Verfeinerung der Class-Seite ist weiter unten abgebildet und wird detailliert erläutert. Entsprechend der hier gewählten Netzdarstellung der Interfaces werden Nachrichten durch die Interfaces aus der Stelle ‘inMsg’ entnommen und in die, von der Methode benötigten, Bestandteile zerlegt. Die bei der Diskussion der Interfaces erwähnten Stellen für die Parameter, den Kontrollfluß und das aufgerufene Objekt sind auf der Klassen-Seite nochmals dargestellt und sind identisch mit denen auf der Interface-Seite. Sie dienen als Eingangsstellen für die Methoden. Auf der Klassen-Seite werden die Methoden als Transitionen dargestellt. Verfeinert werden diese i.a. auf Unterseiten.

Für die new-Methode gibt es keine Verfeinerung auf einer Unterseite. Da eine leere Liste keine Attribute hat, werden bei dem Aufruf der new-Methode auch keine Attribute initialisiert. Nur der Kontrollfluß muß zurückgegeben werden, um anzuzeigen, daß die Initialisierung beendet ist. Der Aufruf reduziert sich auf die Erzeugung der ObjektID durch die Class-Seite.

Für die o.g. add-Methode aus Abbildung 117 zeigt die Netz-Seite #13 in der Abbildung 118 die Verfeinerung. Bei dem Aufruf der add-Methode wird die new-Methode der Klasse ‘NEList’ aufgerufen. Diese Methode wird auf der Class-Seite als Klassenmethode der Klasse ‘NEList’ erkannt und liefert eine neue OID für das neue Listenobjekt. Danach erfolgt die Initialisierung der Attribute der Liste durch den Aufruf der new-Methode bei dem neu erzeugten Objekt der Klasse ‘NEList’. Die Definition dieser Methodensignatur befindet sich, wie oben beschrieben, in dem Interface ‘Pair’ und wird von ‘NEList’ implementiert. Der Rückgabewert der add-Methode entspricht der Definition im

Interface ‘List’.

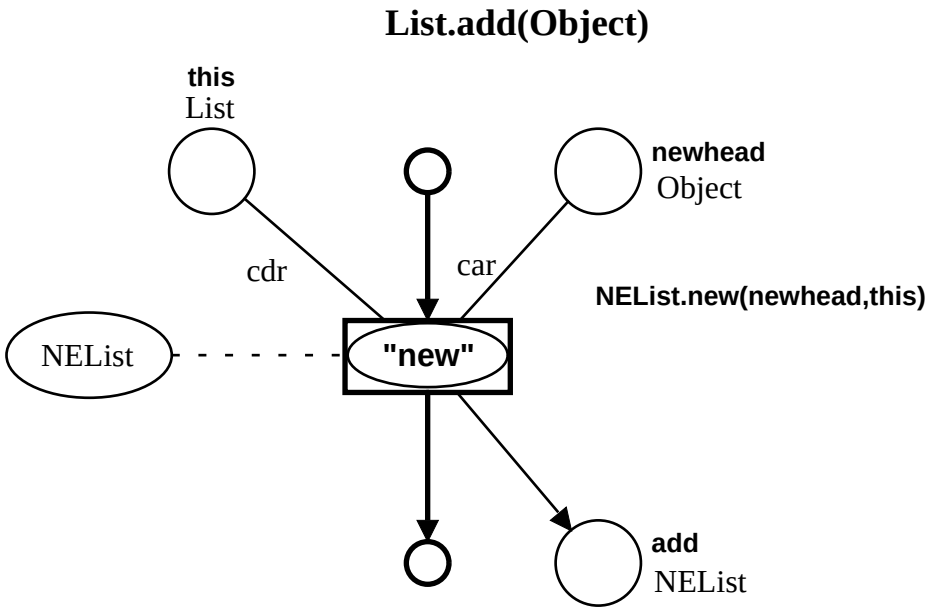


Abbildung 118: Implementation der add-Methode

In der folgenden Abbildung 119 ist die Class-Seite dargestellt. Sie wird von den Klassen-Seiten als Unterseite verwendet. Bei der Entnahme der Nachrichten aus der Stelle ‘CallMsgPool’ wird überprüft, an welche Klasse oder an welche Ausprägung einer Klasse die Nachrichten gerichtet sind. Die obere Transition schaltet, wenn die Klassenmethode ‘new’ aufgerufen wird. Sie erzeugt eine neue ObjektID und ruft die new-Methode dieses neuen Objekts zur Initialisierung eventueller Attributstellen auf. Die untere Transition auf der Class-Seite schaltet bei Aufrufen aller Klassenmethoden, bei denen es sich nicht um die new-Methode handelt. Die mittlere Transition filtert Nachrichten für Klassenausprägungen (Instances) aus der Stelle ‘CallMsgPool’.

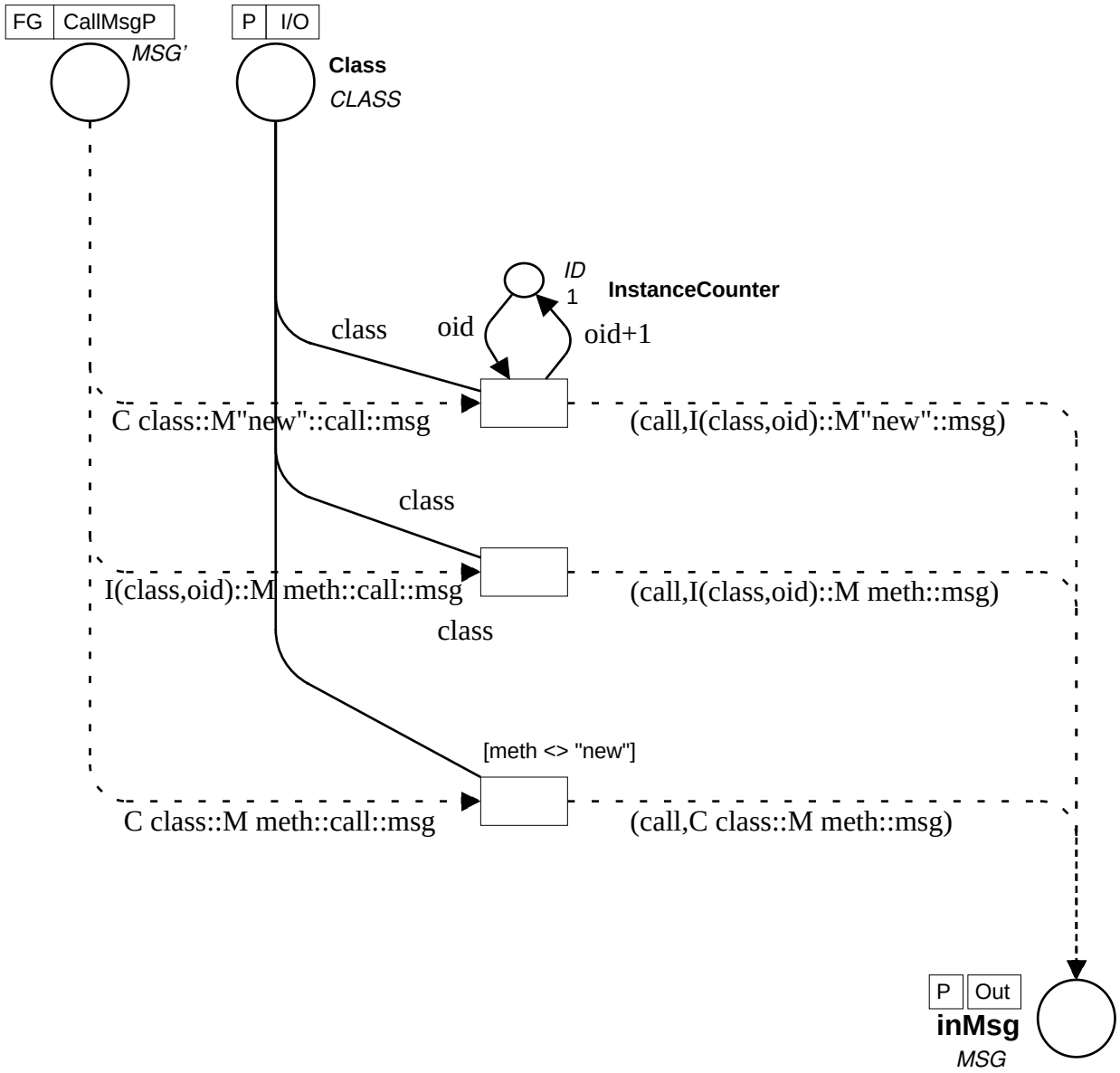


Abbildung 119: Class-Seite

4.8 Werkzeugunterstützung

Zur Entwicklung der objektorientierten Petrinetze wird in dieser Arbeit das Werkzeug ‘Design/CPN – Version 3.02’ eingesetzt. Es bietet gute Möglichkeiten zum Editieren von (gefärbten) Petrinetzen und zur Simulation der modellierten Netze. Es wird seit vielen Jahren am Fachbereich Informatik der Universität Hamburg eingesetzt und hat in [Störrle 98] bei der Untersuchung einer Auswahl von Werkzeugen zur Modellierung von Petrinetzen die beste Beurteilung bekommen. Entwickelt wurde Design/CPN an der Universität in Aarhus und viele Informationen zu diesem Werkzeug findet man unter ‘http://www.daimi.au.dk/designCPN/’.

Leider wird der Entwurf von objektorientierten Petrinetzen nicht unterstützt. Somit müssen alle Netzstrukturen, die automatisch generiert werden könnten, mit allen Petrinetz-Elementen manuell erstellt werden. Dabei handelt es sich um alle Stellen inklusive Name, Farbe und eventuell Initialmarkierung, Transitionen mit Name und eventueller Verfeinerung, sowie die Art der Kanten und deren

Beschriftungen. Anhand eines Klassenausschnitts der später im Fallbeispiel modellierten Klasse ‘Bus’ werden im folgenden die verwendeten Netzteile ausführlich dargestellt und danach mit den ausgeblendeten Elementen und unter Verwendung von abkürzenden Schreibweisen. Dies sind z.B. die Lesekanten, die als ungerichtete Kanten gezeichnet, aber im Petrinetz als zwei gerichtete Kanten in der entgegengesetzten Richtung bei der Simulation ausgeführt werden. Die kleinen Ringe mit den Zahlen in den Stellen wurden bisher nicht erwähnt. Sie dienen zur Positionierung der Marken, die sich während der Simulation in den Stellen befinden können. Die Zahl zeigt die Anzahl der Marken in der Stelle an. Bei Stellen mit einer Initialmarkierung ist die Zahl ‘1’.

Bei den Netzteilen handelt es sich um die Interface-Seite, die Klassen-Seite und eine Methoden-Seite. Die ausführlichen Darstellungen zeigen alle Informationen, die zur Simulation der Petrinetze benötigt werden. Die Abbildungen mit den ausgeblendeten Netzelementen und den abgekürzten Schreibweisen zeigen nur die für die Modellierung interessanten Informationen. Die Interface-Seite und die Klassen-Seite, sowie Teile der Methoden-Seiten könnten aus dem Klassendiagramm automatisch generiert werden. In [Rölke 99] wird die Generierung von Petrinetzen aus Klassendiagrammen betrachtet und teilweise implementiert. Die für diese Arbeit benötigten Generierungsroutinen konnten nicht rechtzeitig fertiggestellt und damit leider nicht eingesetzt werden.

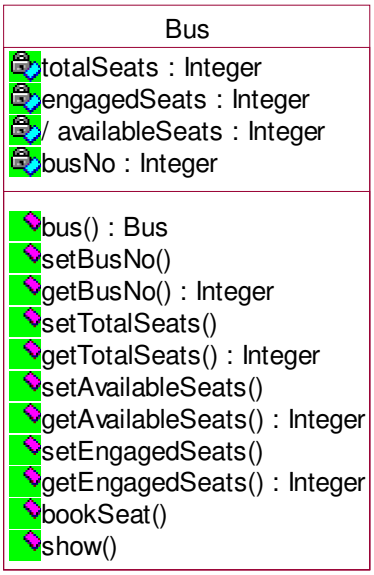


Abbildung 120: Die Bus-Klasse als Klassendiagramm

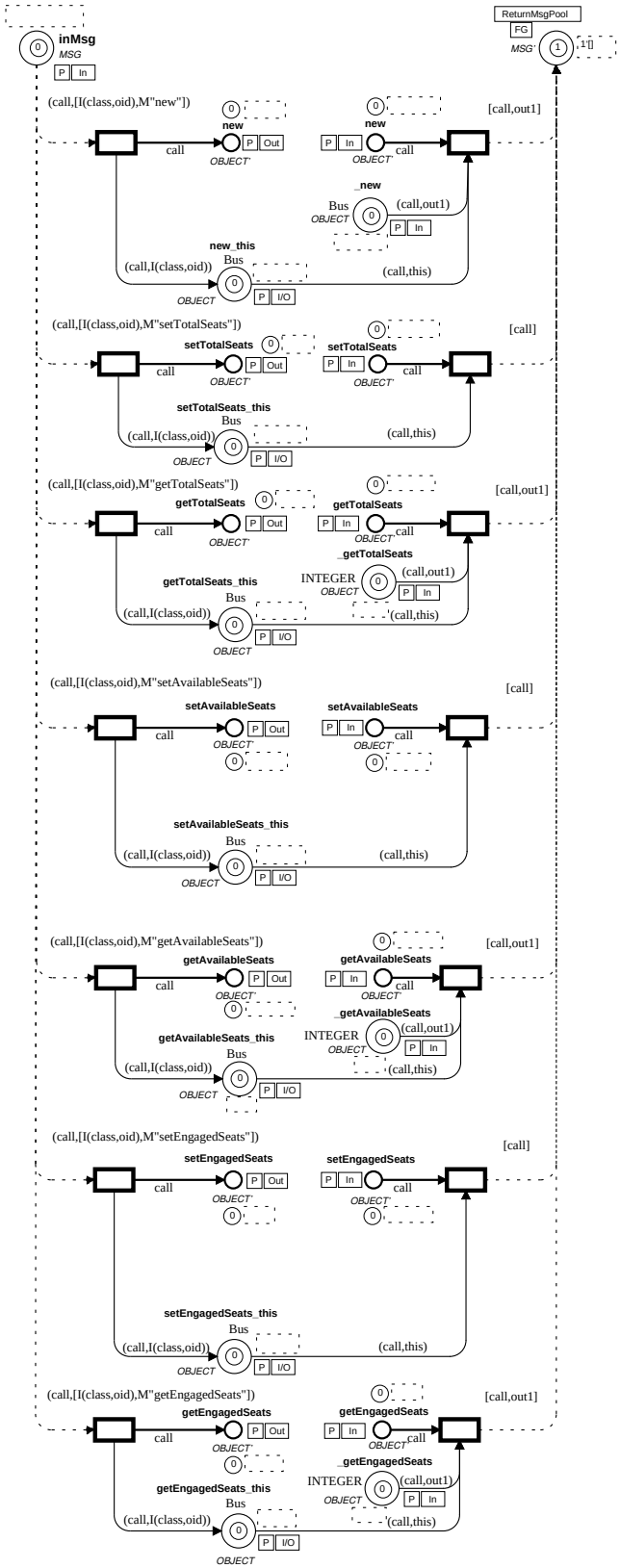


Abbildung 121: Ausführliche Darstellung der Bus-Interface-Seite

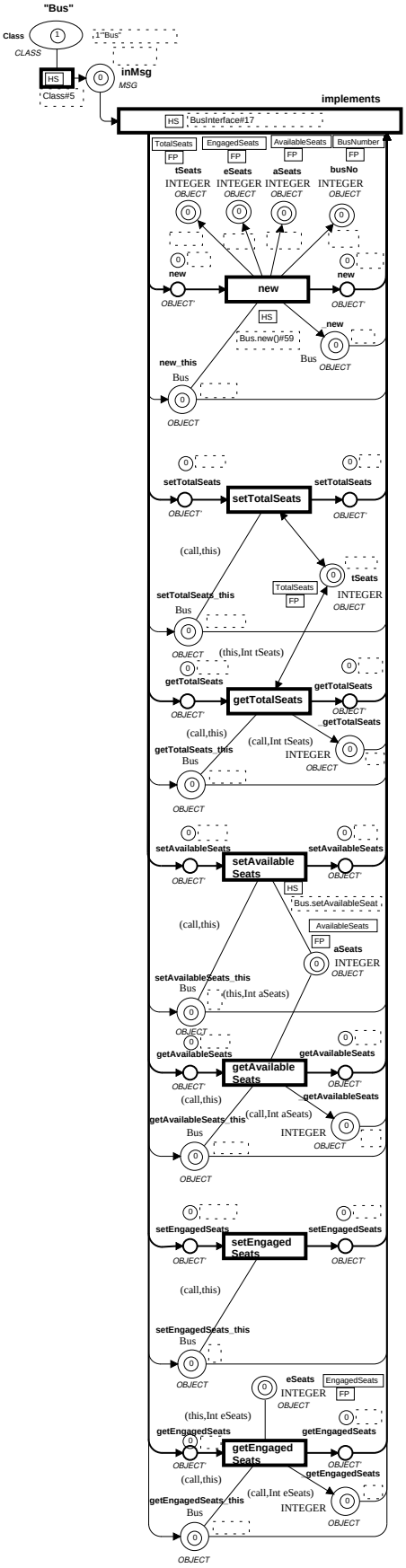


Abbildung 122: Ausführliche Darstellung der Bus-Klassen-Seite

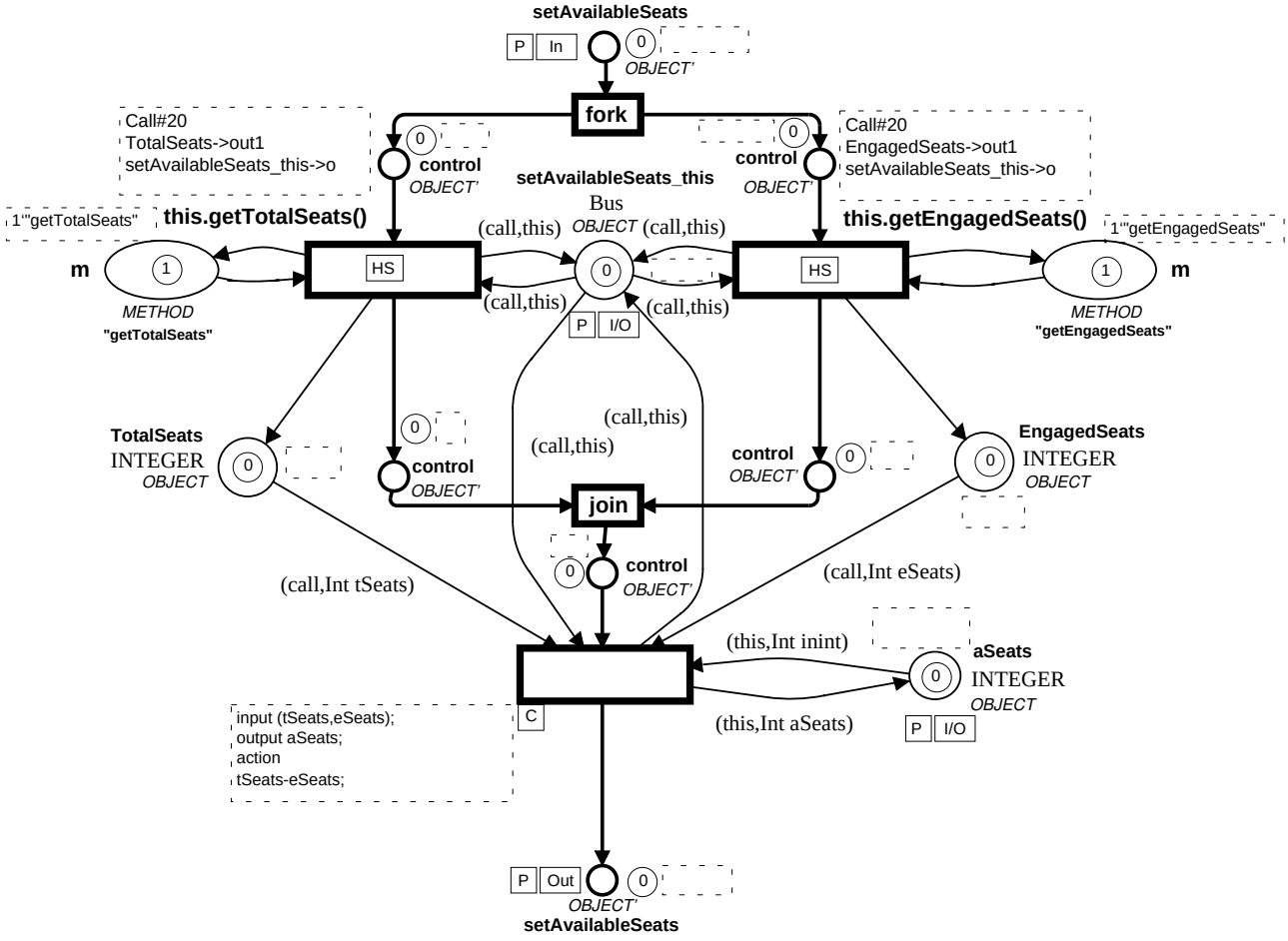


Abbildung 123: Ausführliche Darstellung der Methoden-Seite 'setAvailableSeats()'

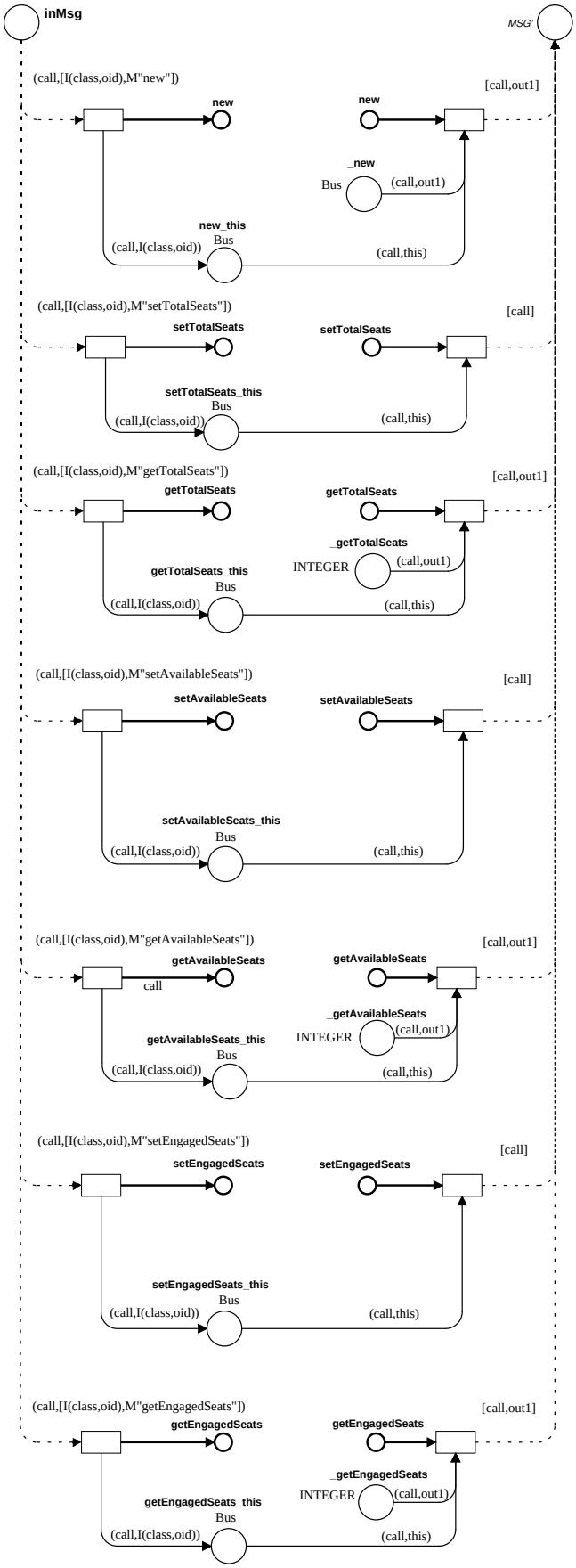


Abbildung 124: Abgekürzte Darstellung der Bus-Interface-Seite

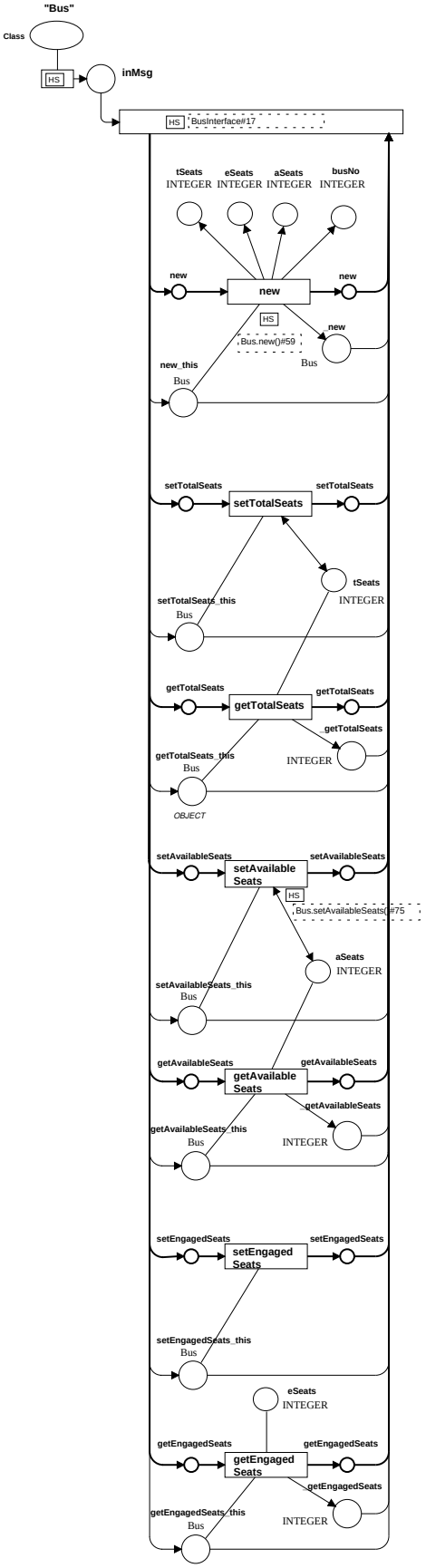


Abbildung 125: Abgekürzte Darstellung der Bus-Klassen-Seite

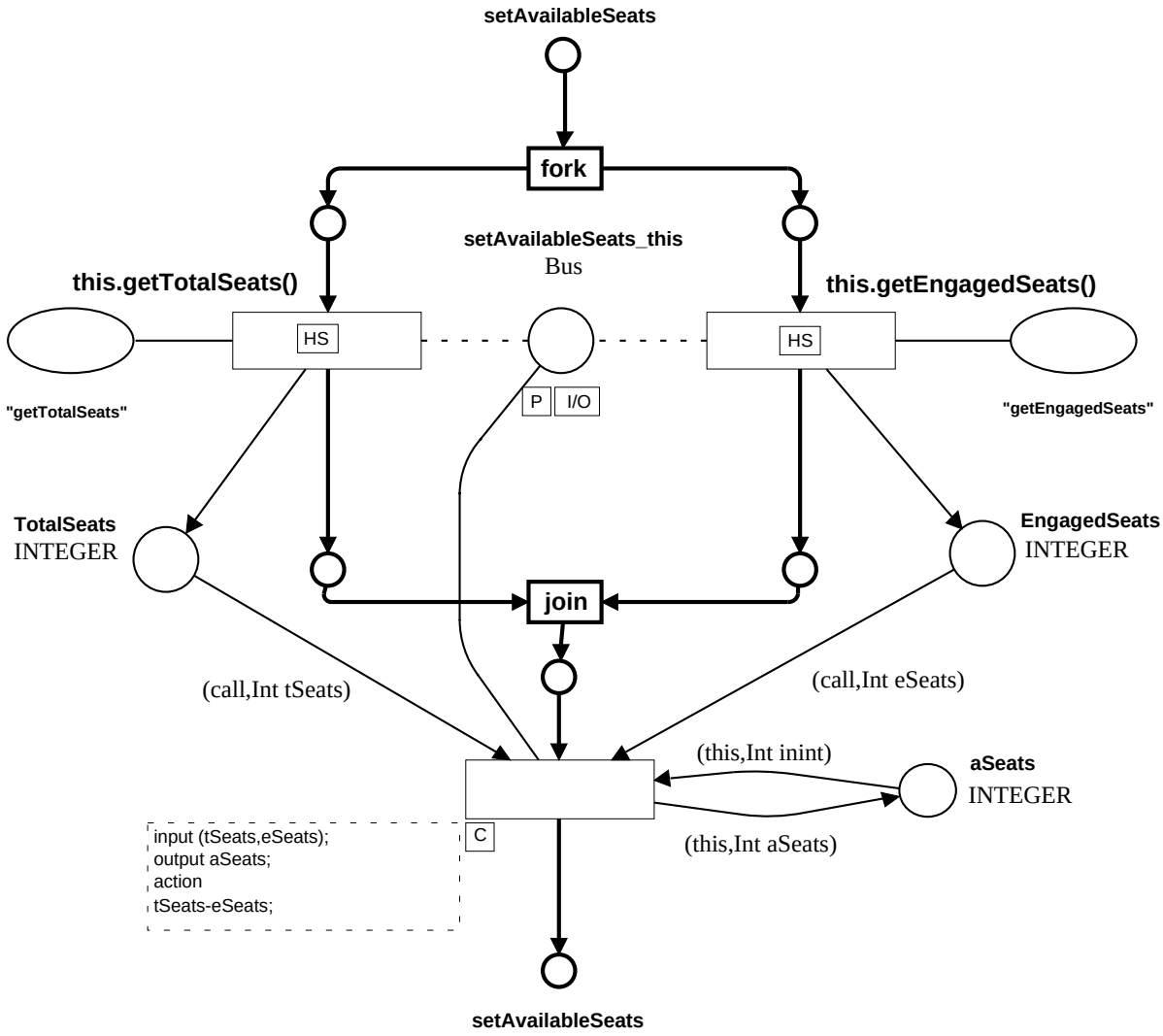


Abbildung 126: Abgekürzte Darstellung der Methoden-Seite 'setAvailableSeats()'

Ein weiterer Aspekt der Werkzeugunterstützung ist das während des Schreibens dieser Arbeit entwickelte und in [KMW 98] vorgestellte Konzept, das die Kommunikation verschiedener Design/CPN-Prozesse über eine Mailbox ermöglicht. Für jeden Prozeß wird ein Messenger zu Beginn der Simulation gestartet. Die Prozesse können auf beliebigen Rechnern laufen, die Zugriff auf die Mailbox haben. Somit kann das gesamte Petrinetz in übersichtliche Teilnetze zerlegt, separat entwikkelt und getestet werden. Steht keine verteilte Entwicklungsumgebung zur Verfügung, so können die Messenger auf einem Rechner laufen und über die Mailbox kommunizieren. Bei der Verwendung mehrerer Design/CPN-Prozesse kann z.Zt. keine gemeinsame Hierarchieseite automatisch erstellt werden und der Austausch von Netzteilen beim Editieren zwischen den verschiedenen Design/CPNs wird nur mäßig unterstützt. Eine Lösung dieser Probleme verspricht der Einsatz von Werkzeugen zur Speicherung von Design/CPN-Diagrammen im Textformat und der Möglichkeit diese in bereits bestehende Diagramme zu integrieren. In [Maier 98] und [Lyngsø 98] werden Ansätze zur Realisierung der Speicherung von Design/CPN-Diagrammen im Textformat vorgestellt, konnten aber für diese Arbeit noch nicht praktisch eingesetzt werden.

Das Mailbox-Konzept ermöglicht die Kommunikation zwischen Petrinetz-Objekten und Java-Objekten. Die grafische Benutzeroberfläche eines modellierten Systems könnte also problemlos in Java programmiert werden. Die Entwicklung einer solchen grafischen Oberfläche ist hier nicht das Ziel der Modellierung und somit im Rahmen dieser Arbeit nicht vorgesehen.

4.9 Einbettung von OOCPN in die Analyse

Ein Vorgehen zum Einsatz der objektorientierten Petrinetze zur Spezifikation von Systemen wird in [Moldt 96] vorgeschlagen. Vorausgesetzt wird ein Werkzeug zur Entwicklung eines Modells des Systems mit den aus anderen objektorientierten Analyseansätzen bekannten Techniken zur Modellierung von statischen, dynamischen und funktionalen Systemaspekten. Bei dem vorausgesetzten Werkzeug können als Modellierungstechniken von dem Entwickler Klassendiagramme, Statecharts und Datenflußdiagramme eingesetzt werden. Diese Diagramme werden von dem Werkzeug in Petrinetze übersetzt und diese können dann simuliert und damit das modellierte System validiert werden. Als Ergänzung zum statischen Modell findet eine funktionale und zustandsorientierte Zerlegung statt. Durch die Simulierbarkeit eignen sich die Petrinetze zum Prototyping. Dieses wird mit einem iterativen Vorgehen kombiniert.

Bei dem Entwurf der Spezifikation werden verschiedene Tätigkeiten durchgeführt. Es erfolgt eine Festlegung des Kontextes und der Schnittstelle nach außen, das statische, das dynamische und das funktionale Modell wird entwickelt und die Modelle werden integriert. Durch die klare Trennung von statischem, dynamischem und funktionalem System kann gezielt auf die verschiedenen Aspekte der Systemschnittstelle eingegangen werden. Dabei enthält besonders das dynamische Modell Informationen über die Interaktion des Systems mit der Umwelt und das funktionale Modell beschreibt die Daten, die ins System gelangen bzw. vom System geliefert werden. Zur Veranschaulichung des Vorgehens und der resultierenden Netze wird hier ein Ausschnitt der Bus-Klasse aus dem später modellierten Fallbeispiel exemplarisch betrachtet. Das (hier sehr einfache) Klassendiagramm und damit die statische Sicht zeigt die Abbildung 127. Die Abbildung 128 zeigt das Statechart-Diagramm für diese Klasse.

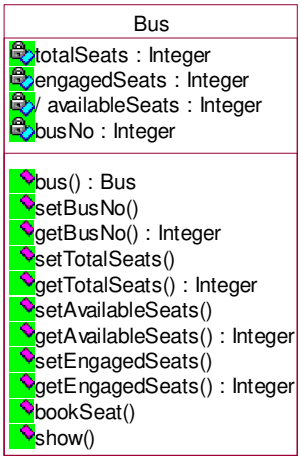


Abbildung 127: Ausschnitt aus der Bus-Klasse

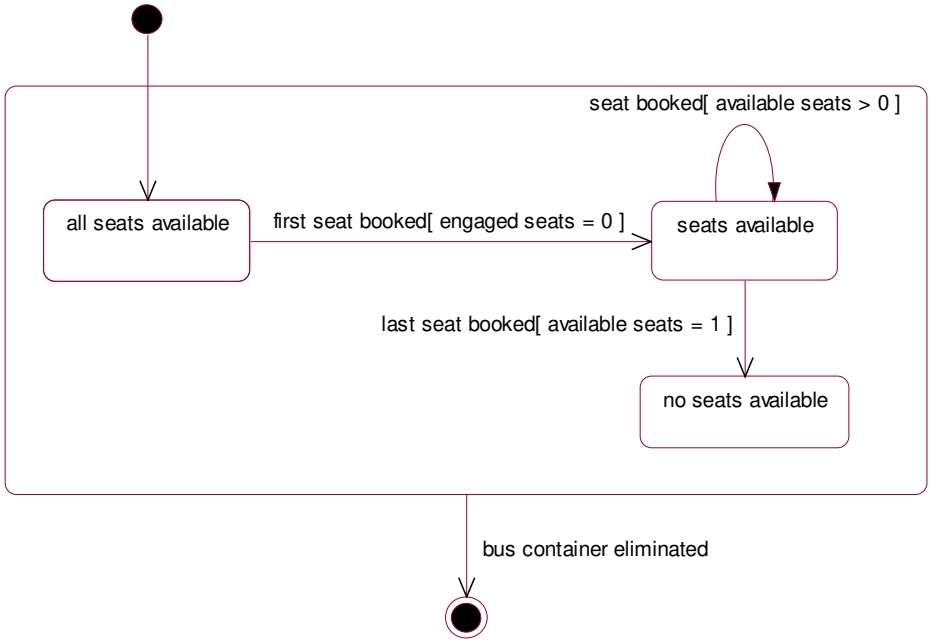


Abbildung 128: Statechart der Bus-Klasse

In der Abbildung 129 ist das Zustandsobjekt dargestellt. Nach dem ersten Aufruf der Methode ‘f_book()’ im Funktionsobjekt befindet sich das Bus-Objekt in dem Zustand, in dem noch Sitzplätze verfügbar aber nicht mehr alle frei sind. Bei der Buchung des letzten Sitzplatzes wechselt das Bus-Objekt in den Zustand, der anzeigt, daß alle Plätze belegt sind.

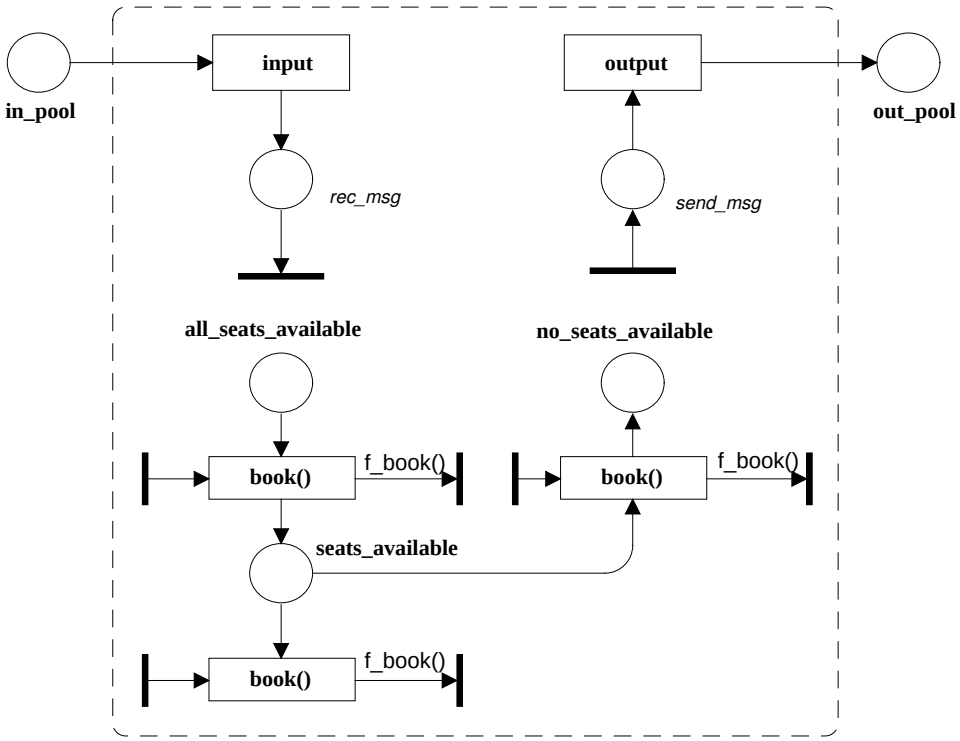


Abbildung 129: Statechart Bus als Zustandsobjekt

Die Abbildung 130 zeigt das Funktionsobjekt. Von den spezifizierten Methoden wird nur die Methode ‘f_book()’ dargestellt, weil nur diese einen Zustandswechsel bewirkt. Sie wird von dem dynamischen Zustandsobjekt aufgerufen. Durch die getrennte Darstellung der unterschiedlichen Aspekte kann sich der Entwickler bei der Betrachtung eines Netzes auf eine Sicht konzentrieren. Als weiterer Schritt können die Petrinetze integriert werden. Damit werden dann die statischen, die dynamischen und die funktionalen Aspekte durch eine Technik in einem Modell beschrieben. Die Darstellung des integrierten Modells zeigt das Klassendiagramm aus der Abbildung 131. Die gerichteten Kanten zeigen die Richtung der Aufrufe.

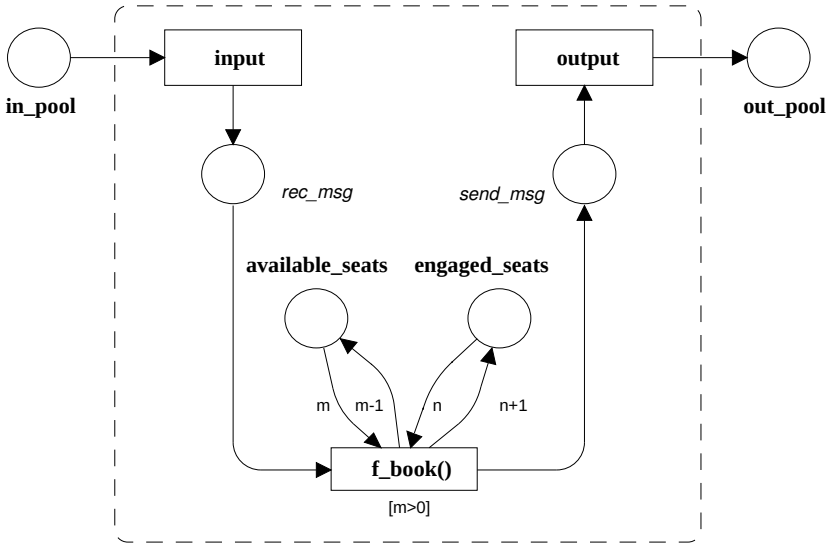


Abbildung 130: Funktionsobjekt Bus

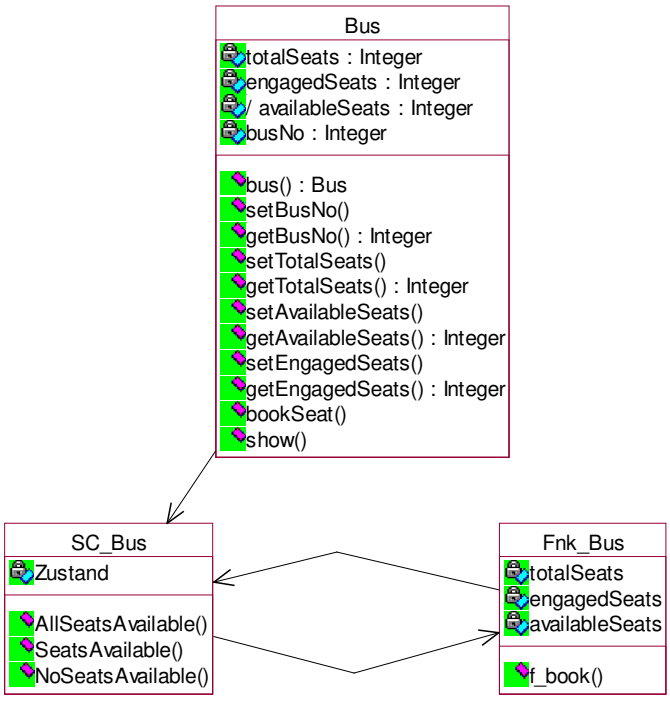


Abbildung 131: Integration des Zustands- und FunktionsObjekts

In dieser Arbeit wird ein anderes Vorgehen gewählt, weil das in [Moldt 96] erwähnte, benötigte Werkzeug nicht zur Verfügung steht. Bei dem hier vorgestellten Vorgehen werden auf der Basis von UML-Diagrammen der objektorientierten Analyse ausführbare Petrinetze entworfen. Durch Instanziierung der Klassennetze und deren Simulation werden die modellierten Systemfunktionen getestet. Änderungen werden direkt im Petrinetz vorgenommen und in die UML-Diagramme übernommen. Werden Änderungen an den UML-Diagrammen vorgenommen, so werden diese Änderungen in das Petrinetz integriert. Die Petrinetz-Prototypen werden durch Iteration über die Entwicklungsphasen Analyse, Design, Implementation und Testen im Sinne der ausführbaren Systemspezifikation erstellt.

Die Abbildung 132 zeigt alle Phasen bei der Entwicklung einer ausführbaren Systemspezifikation mit objektorientierten Petrinetzen und der Verwendung von UML-Diagrammen zur Systemanalyse. Durch die hervorgehobenen Stellen und Kanten wird der Kontrollfluß dargestellt. Die großen Transitionen sind Entwicklungsaktivitäten, die Daten verwenden oder erzeugen, die in den Stellen als Marken vorhanden sein können. Die mit ‘FP’ gekennzeichneten Stellen sind Seiten-Fusions-Stellen, d.h. alle Stellen, die durch einen gemeinsamen Fusionsnamen bezeichnet werden, können als eine Stelle betrachtet werden. Der Einsatz von Fusions-Stellen vermeidet Kantenüberschneidungen. Kanten ohne Pfeilspitzen sind Lesekanten, und beim Schalten einer Transition werden keine Marken von der mit dieser Kantenart verbundenen Stelle abgezogen. Die Kanten mit Pfeilspitzen an beiden Enden bedeuten, daß eine Marke aus einer Stelle entnommen, jedoch nicht verändert wird. Der Unterschied besteht darin, daß bei Lesekanten konzeptuell nebenläufige Markenverwendung möglich ist, während bei der anderen Modellierung Marken entnommen werden und Nebenläufigkeit dadurch ausgeschlossen wird.

Zu Beginn befindet sich in der Stelle ‘Start’ eine Kontrollmarke. Ist eine textuelle Systembeschreibung vorhanden, so ist die erste Transition aktiviert. Das Schalten dieser Transition repräsentiert die Erzeugung eines ersten Use-Case-Diagramms unter Verwendung der textuellen Systembeschreibung. Als nächstes wird ein erstes Klassendiagramm entworfen. Hierzu wird die textuelle Beschreibung und das Use-Case-Diagramm betrachtet. In dem nächsten Schritt werden zu dem Klassendiagramm Interaktionsdiagramme entworfen. In dieser Abbildung repräsentiert die Stelle ‘Interaktionsdiagramm’ die Menge aller zum Klassendiagramm korrespondierenden Interaktionsdiagramme. Diese ersten drei Schritte beschränken sich auf die Analysephase der Software-Entwicklung. Das Klassendiagramm und die Interaktionsdiagramme dienen als Grundlage zur Modellierung eines ersten Petrinetzes. Um dieses Petrinetz auszuführen, müssen einige Entwurfsentscheidungen getroffen und Methoden implementiert und getestet werden. D.h. bei der Netzmodellierung werden die Entwicklungsphasen Analyse, Design, Implementation und Testen, im Kontext der Entwicklung einer Systemspezifikation, durchlaufen. Neben dem ersten Petrinetz-Prototyp in der Stelle ‘Petrinetz’ als Ergebnis, wird die Iterationsnummer mit ‘1’ initialisiert. Bei den darauf folgenden Iterationen werden jeweils alle Diagramme betrachtet. Alle Diagramme einer Iteration müssen dabei parallel so bearbeitet werden, daß sie die Konsistenz des Modells bewahren. Durch Prototyping, d.h. die Weiterentwicklung des Petrinetzes, beeinflussen in erster Linie die Ergebnisse dieses Modellierungsprozesses die anderen Diagramme. Nach einer beliebigen Anzahl von Iterationen wird bei der Erfüllung eines gewählten Abbruchkriteriums der Entwicklungsprozeß beendet. Das Ergebnis ist eine ausführbare Systemspezifikation, bestehend aus dem letzten Petrinetz-Prototyp und einem Dokument mit Erläuterungen zu diesem. Ein Beispiel wird an dieser Stelle nicht betrachtet, weil das Vorgehen im folgenden Kapitel für das Fallbeispiel ausführlich dokumentiert wird.

Vorgehen bei der Entwicklung einer ausfuehrbaren Systemspezifikation mit UML und OOCPN

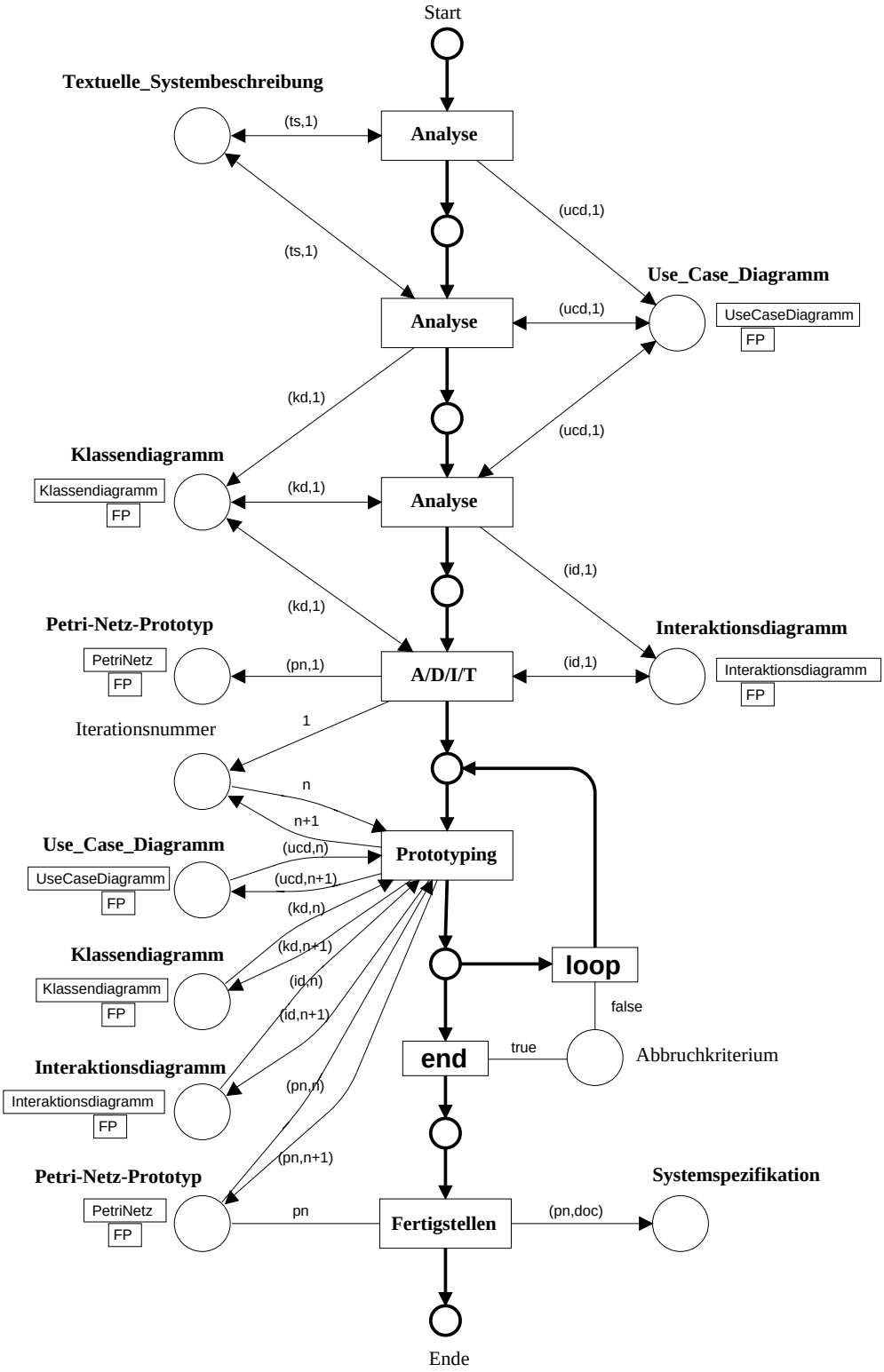


Abbildung 132: Vorgehen bei der Modellierung

5 EINSATZ VON OOCPN ZUR SPEZIFIKATION VON MINITOURS

In diesem Kapitel wird das Reiseunternehmens ‘Minitours’ mit objektorientierten gefärbten Petrinetzen (OOCPN) spezifiziert. Zuvor wird der zur Modellierung gewählte Petrinetz-Ansatz an einer Klasse des Fallbeispiels erläutert. Auf der Grundlage der UML-Diagramme werden Klassen-Netze entsprechend dem von mir im vorherigen Kapitel vorgeschlagenen Vorgehen entwickelt. Anschließend werden Modellierungsentscheidungen diskutiert, die im unmittelbaren Zusammenhang mit dem Einsatz der Petrinetze zur objektorientierten Systemanalyse stehen. In einem eigenen Abschnitt wird die Netzarchitektur betrachtet. Darauf folgen Erläuterungen zur Simulation der ausführbaren Systemspezifikation. Zum Abschluß dieses Kapitels werden die Ergebnisse dieser Arbeit zusammengefaßt.

5.1 Verwendete Petrinetz-Notation

Zur Modellierung des Reiseunternehmens ‘Minitours’ wird hier der Interface-Ansatz verwendet. Zu einer Klasse des UML-Klassendiagrammes wird ein Interface als Teil-Petrinetz kreiert, das die Methodensignaturen spezifiziert. Nachrichten an ein Objekt der entsprechenden Klasse werden für die aufgerufene Methode aufbereitet. Ein weiteres Teilnetz realisiert die Klasse, die das Interface implementiert. In diesem Netz werden die Instanzvariablen als Stellen modelliert und die Methoden als Transitionen. Auf Unterseiten sind die Methoden als Verfeinerungen (weitere Teilnetze) implementiert.

Der Interface-Ansatz wird hier gewählt, weil er dem in der objektorientierten Programmiersprache Java realisierten Interfacekonzept entspricht und eine technische Realisierung zur Simulation von Frank Wienberg bereitgestellt wurde. Da es sich bei Java um eine Programmiersprache handelt, die die objektorientierten Konzepte sauber umsetzt, lassen sich diese Vorteile auch bei der Modellierung der Petrinetze nutzen. Als Beispiel für diesen sauberen Entwurf wird an dieser Stelle exemplarisch die Klasse TravelSchemeContainer betrachtet. Die Modellierung weiterer Klassen des Reiseunternehmens wird weiter unten in diesem Kapitel diskutiert.

In der Abbildung 133 ist das Interface für die Klasse ‘TravelSchemeContainer’ dargestellt. Die im Klassendiagramm mit ‘travelSchemeContainer()’ benannte Konstruktormethode heißt hier ‘new’. Sie erhält keine Parameter und liefert einen Wert zurück. Bei diesem Rückgabewert handelt es sich um die Referenz des neu erzeugten Objekts. Um diesen Wert zurückgeben zu können, muß die new-Methode diesen als eine Art Parameter erhalten. Es handelt sich dabei aber nicht um einen Parameter für die Methode im ursprünglichen Sinne, sondern um eine technisch notwendige Information, da bei dieser Art Petrinetze mehrere Netze zusammengefaltet sein können. Aus diesem Grund wurde die new-Methode als parameterlos beschrieben. Weitere vom Interface spezifizierte Methodensignaturen sind ‘createTravelScheme()’, ‘storeTravelScheme()’ und ‘browse()’. Daß auch diese Methoden die Selbstreferenz benötigen wird bei der Vorstellung der Methodenimplementationen deutlich.

Nachdem eine Nachricht für eine Methode eines Objekts aus der Stelle ‘inMsg’ durch eine Transition auf der Interface-Seite entnommen wurde, stehen die von der Methode benötigten Bestandteile der Nachricht in den Eingangsstellen der Methodentransition auf der Klassen-Seite durch die Verwendung von Fusions-Stellen zur Verfügung. In der Abbildung 134 ist die Klassen-Seite für die Klasse ‘TravelSchemeContainer’ dargestellt.

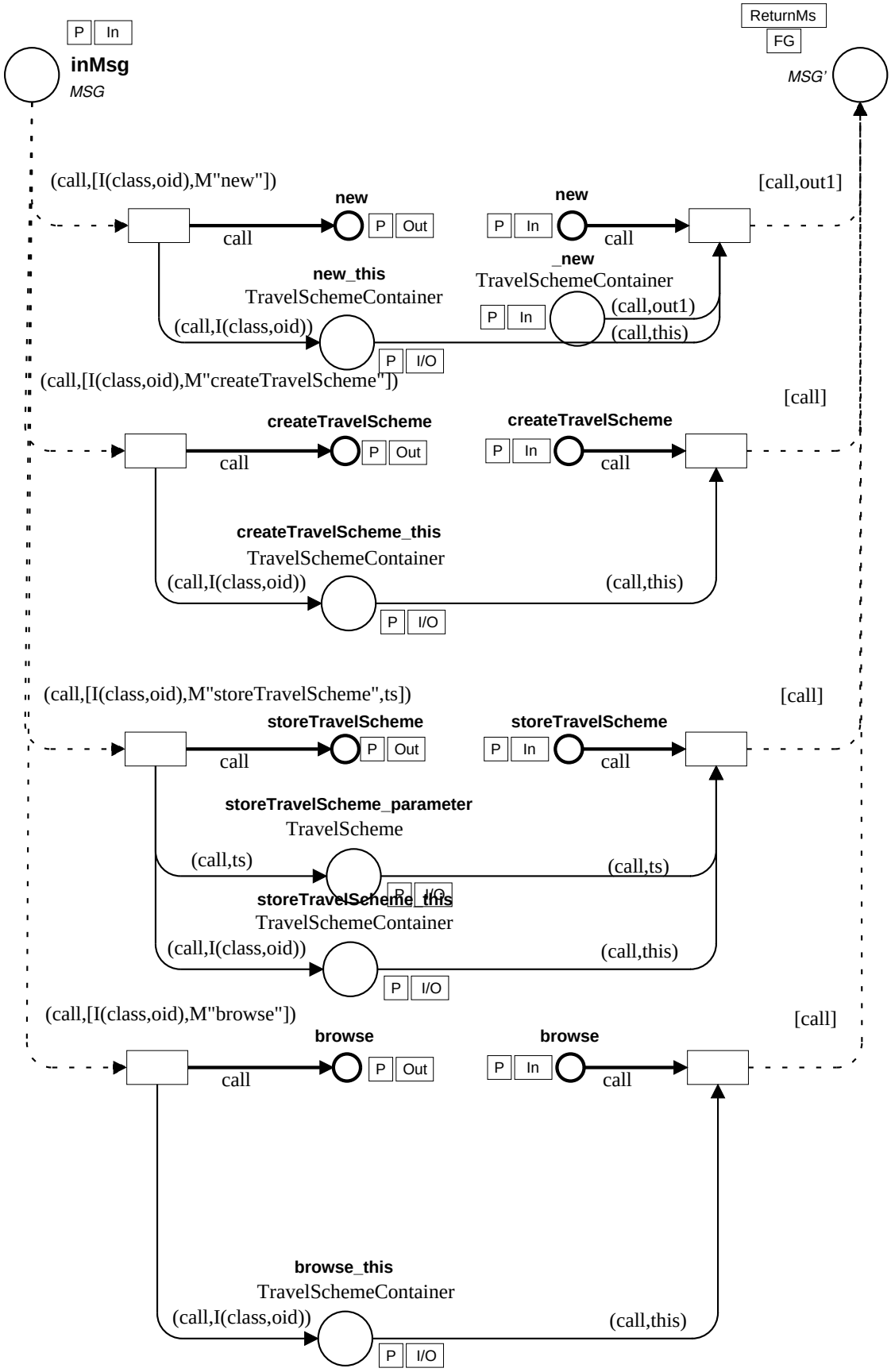


Abbildung 133: Interface-Seite der Klasse TravelSchemeContainer

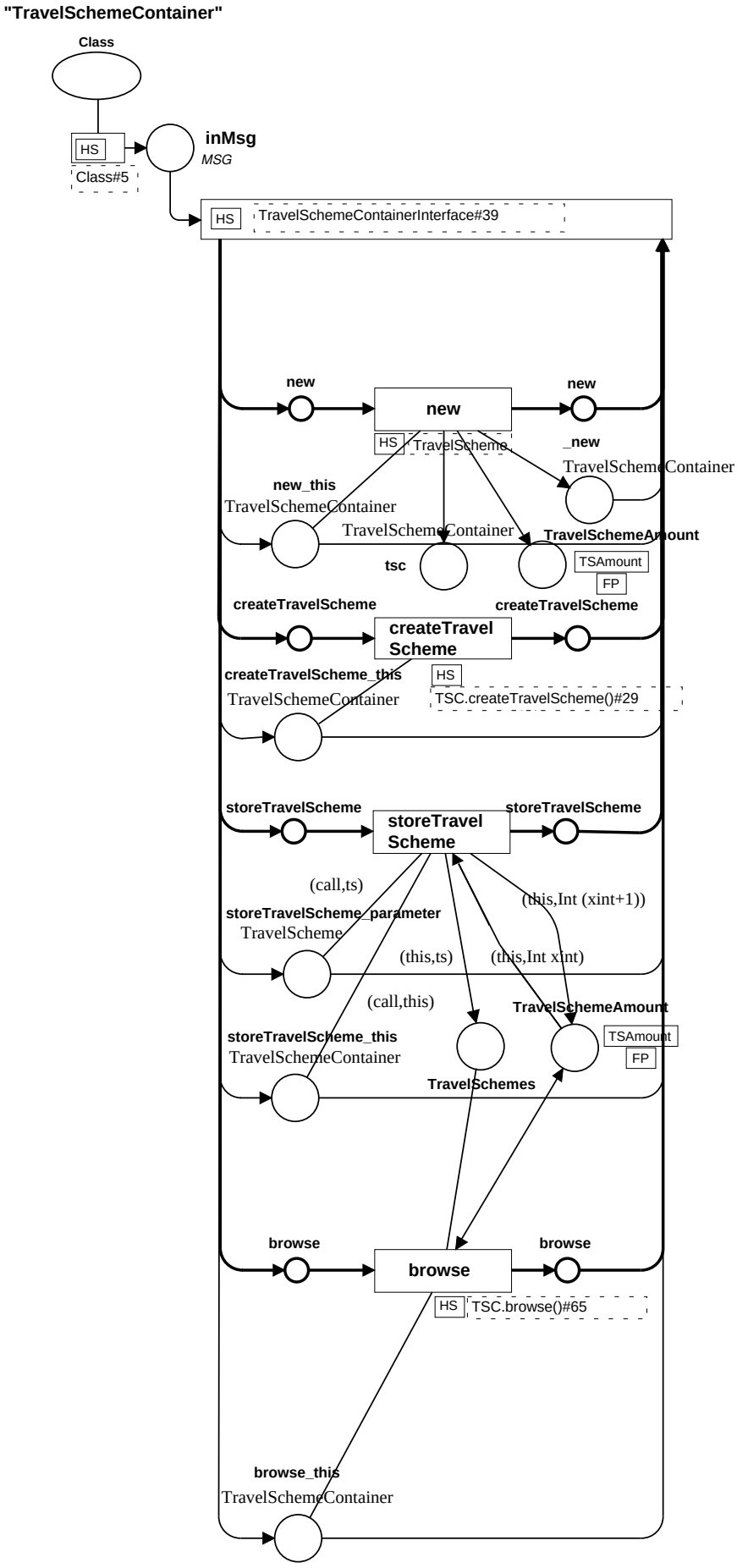


Abbildung 134: Klassen-Seite der Klasse TravelSchemeContainer

Die Methoden sind i.a. auf Unterseiten als Verfeinerung modelliert, um die Übersichtlichkeit der Klassen-Seite zu erhalten. Bei der Methode storeTravelScheme ist dies nicht der Fall, da die Funktionalität durch wenige Kantenbeschriftungen realisiert wird. Die Verfeinerung einer Methode kann beliebig komplex gestaltet werden und weitere Unterseiten verwenden. Als Beispiel wird hier die Methode ‘createTravelScheme()’ in der Abbildung 135 betrachtet. Der Kontrollfluß läßt erkennen, daß die Methoden ‘TravelScheme.new()’ und ‘this.storeTravelScheme(ts) ’ sequentiell aufgerufen werden. Die erste Transition generiert eine Nachricht mit dem Aufruf der new-Methode bei der Klasse ‘TravelScheme’. Erkennbar ist dies durch die Klasse als Empfänger der Nachricht in der Stelle ‘o’ mit der Initialmarkierung ‘(Void,C“TravelScheme“)’ und der Stelle ‘m’ mit der Initialmarkierung ‘“new“’. Die Beschriftung ‘HS’ der Transition deutet auf eine weitere verwendete Unterseite hin. Es handelt sich um die Call-Seite zur Erzeugung der erwähnten Nachricht an die Klasse ‘TravelScheme’. Die Abarbeitung der new-Methode der Klasse ‘TravelScheme’ liefert einen Wert zurück, nämlich eine Referenz auf das erzeugte Objekt vom Typ ‘TravelScheme’. Diese Referenz wird von der Methode ‘createTravelScheme()’ bei der zweiten Transition benötigt. Sie wird dort als Parameter ‘ts’ in einer Nachricht mit dem Aufruf der Methode ‘storeTravelScheme(ts)’ bei sich selbst übergeben. Dies ist ein Beispiel für die Verwendung der Selbstreferenz. Auch hier erfolgt die Generierung einer Nachricht auf der Unterseite Call. Die Call-Seite kann beliebig oft als Unterseite verwendet werden, da die generierten Nachrichten durch unterschiedliche Aufrufidentifikationen gekennzeichnet sind.

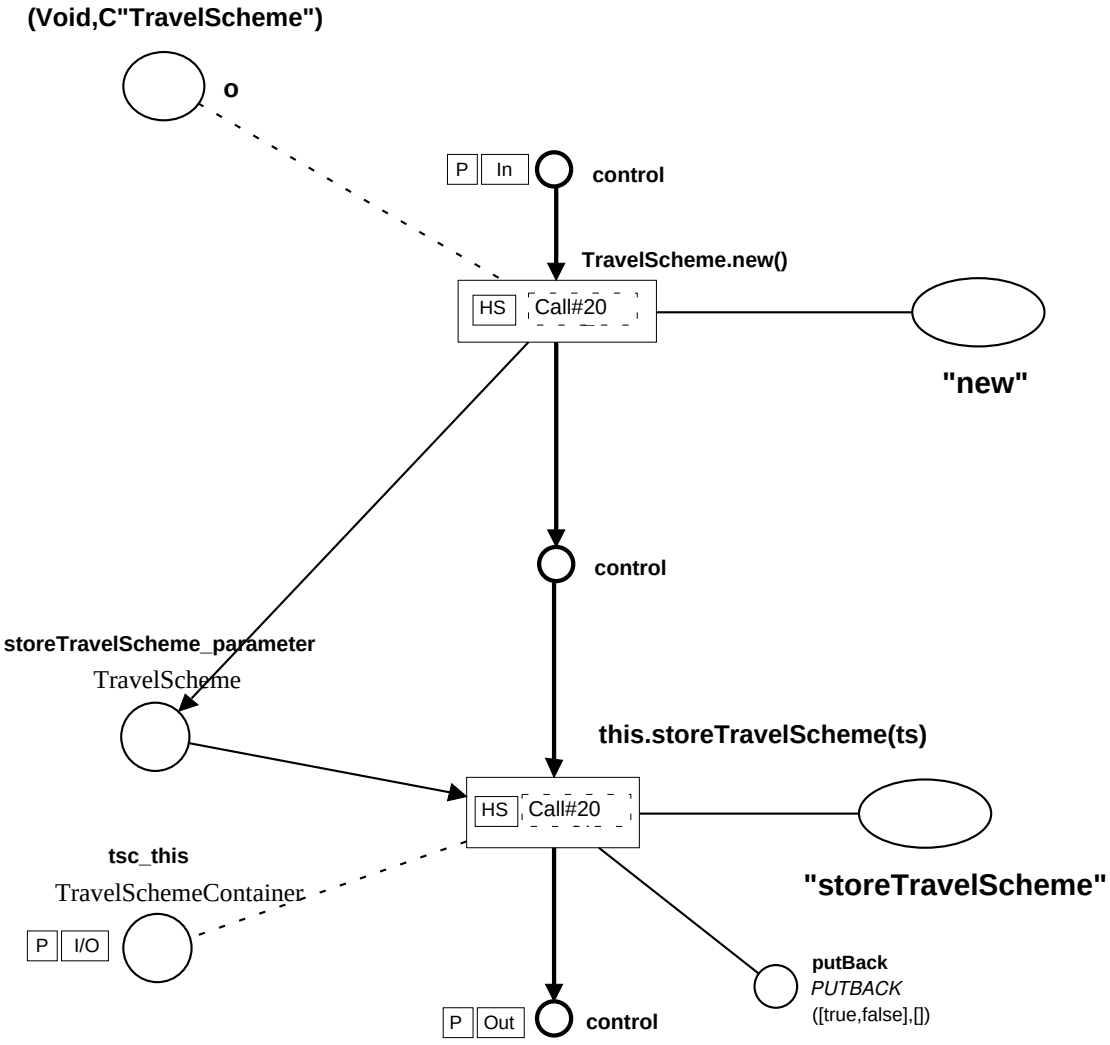


Abbildung 135: Verfeinerung der Methode createTravelScheme

Bei der zweiten Transition wird zusätzlich die Stelle ‘putBack’ verwendet. Diese wird hier benötigt, da per Voreinstellung die Marken in die Stellen, die mit den Stellen der Call-Seite verschmolzen sind, zurückgelegt werden. Dies ist für die Marke mit dem Methodennamen in der Stelle ‘m’ gewünscht, weil dieser bei dem nächsten Aufruf der Transition benötigt wird. Auch die Selbstreferenz in der Stelle ‘tsc_this’ muß zurückgelegt werden, weil diese Marke nach der Abarbeitung der Methode ‘createTravelScheme()’ auf der Oberseite (Klassen-Seite) aus der Stelle abgezogen wird. Bei der Marke in der Stelle ‘storeTravelScheme_parameter’ würde ein Zurücklegen bedeuten, daß dort Marken ungewollt akkumuliert werden würden, daß dann ein nichtdeterministisches und vor allem falsches Verhalten der Methode ‘createTravelScheme()’ zur Folge hätte. Die Initialmarkierung der Stelle ‘putBack’ verhindert das Zurücklegen der Marke aus der Parameter-Stelle durch den false-Eintrag in der ersten Liste an der zweiten Position. Die erste Positon bezieht sich auf die Marke in der Stelle ‘o’ und die zweite auf die Marke in der Stelle mit dem ertsen Parameter, hier ist es die Stelle ‘storeTravelScheme_parameter’. Dadurch wird die Marke in der Stelle ‘tsc_this’, die mit der Stelle ‘o’ auf der Call-Seite verschmolzen ist, nach der Konstruktion der Nachricht, mit dem Aufruf der Methode ‘storeTravelScheme()’ bei dem aufrufenden Objekt selbst, zurückgelegt. Die Parametermarke wird nicht zurückgelegt.

Entsprechend der Soundness-Eigenschaft aus [v.d.Aalst 97] ist die einzige, nach der Abarbeitung einer Methode markierte Kontrollstelle, die mit ‘P Out’ gekennzeichnete Kontrollstelle. Eine Verletzung dieser Eigenschaft kann zu unvorhersehbaren Fehlfunktionen der Petrinetze führen.

Die Entscheidung, den Interface-Ansatz zu verwenden, schließt die Integration anderer Ansätze nicht aus. Entsprechend der Kommunikation zwischen objektorientierten Petrinetzen und Javaobjekten in [KMW 98], können Nachrichten so transformiert werden, daß Objektnetze unterschiedlicher Ansätze miteinander kommunizieren können. Diese Nachrichtentransformation zu realisieren ist nicht das Ziel dieser Arbeit. Da es zur Zeit noch keine Software gibt, die den Entwurf objektorientierter Petrinetze unterstützt und die Nachrichtentransformation noch nicht implementiert ist, wird in dieser Arbeit das Fallbeispiel nur mit dem gewählten Interface-Ansatz modelliert. Für die Erprobung einer Vorgehensweise bei der Entwicklung der Netze zur Spezifikation des Reiseunternehmens bedeutet dies, daß keine Aussage darüber gemacht werden kann, ob sich diese auch für die anderen Realisierungen eignet.

5.2 Vorgehen

Die Entwicklung der als objektorientiertes Petrinetz modellierten, ausführbaren Spezifikation des Reiseunternehmens beginnt mit der Systemanalyse unter Verwendung einiger Techniken der Unified Modeling Language (UML). Bei der hier gewählten Vorgehensweise werden zuerst Use-Cases definiert und zu diesen ein Klassendiagramm und Interaktionsdiagramme entworfen. Auf dieser Grundlage werden Klassen im Sinne des evolutionären Prototyping als objektorientierte Petrinetze modelliert und die in den Interaktionsdiagrammen dargestellten Szenarien getestet. Aufgrund der Ausführbarkeit des Systems umfaßt der Entwicklungsprozeß der ausführbaren Spezifikation nicht nur den Bereich der Systemanalyse, sondern im Kontext dieser Arbeit auch Design, Implementation und Testen. Die Aktivitäten dieser Phasen werden iterativ ausgeführt, d.h. sie werden zirkulär durchlaufen. Neue Erkenntnisse in einer Phase wirken sich i.a. auf andere Phasen aus. Zur Veranschaulichung des Vorgehens dient die Abbildung 132 aus Kapitel 4.9.

Im folgenden wird die Entwicklung der Minitours-Spezifikation vorgestellt. Aufgrund des Umfangs werden an dieser Stelle nicht alle Klassen diskutiert und nicht alle Szenarien betrachtet. Exemplarisch für die Vorgehensweise erfolgt hier die Betrachtung der Entwicklung der Klassen für die Realisierung der Szenarien des Systemstarts, des Anmeldens eines Managers und der Erzeugung eines Reise-

schemas. Das Use-Case-Diagramm ist in der Abbildung 136 dargestellt.

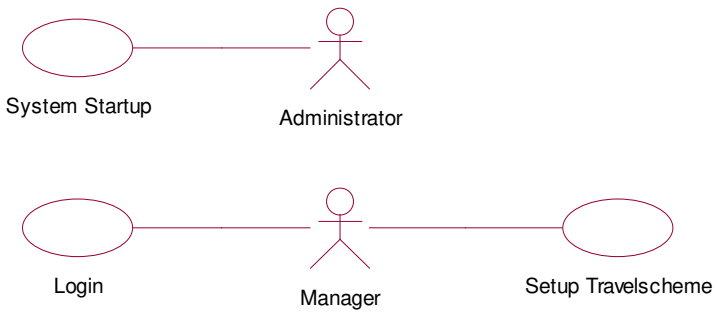


Abbildung 136: Use-Case-Diagramm

Die Abbildung 137 zeigt den hier relevanten Ausschnitt des Minitours-Klassendiagramms. Die Sequenzdiagramme werden in diesem Abschnitt nicht alle noch einmal gezeigt, da sie bereits bei der Spezifikation von Minitours mit UML abgebildet wurden. Nur das Diagramm des Systemstarts wird hier in der Abbildung 138 wiederholt dargestellt, um die für den im folgenden vorgestellten modellierten Teil des Fallbeispiels benötigten Diagramme an dieser Stelle präsent zu haben.

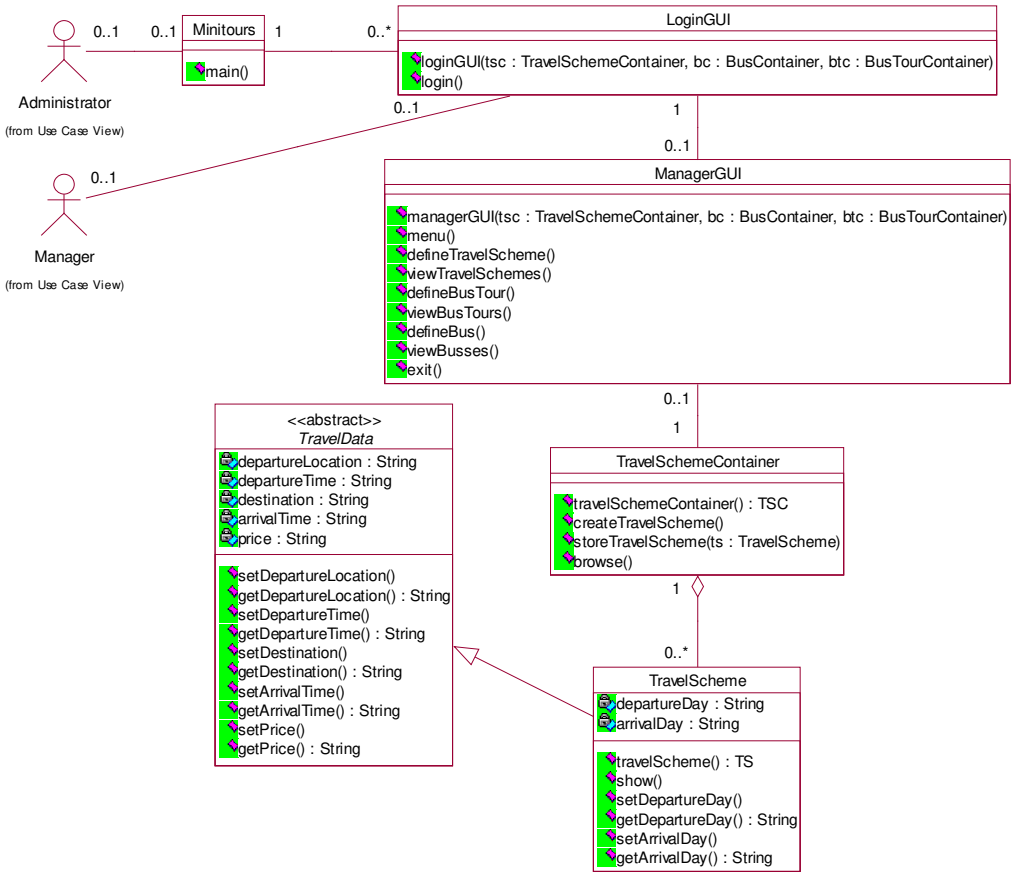


Abbildung 137: Klassendiagramm

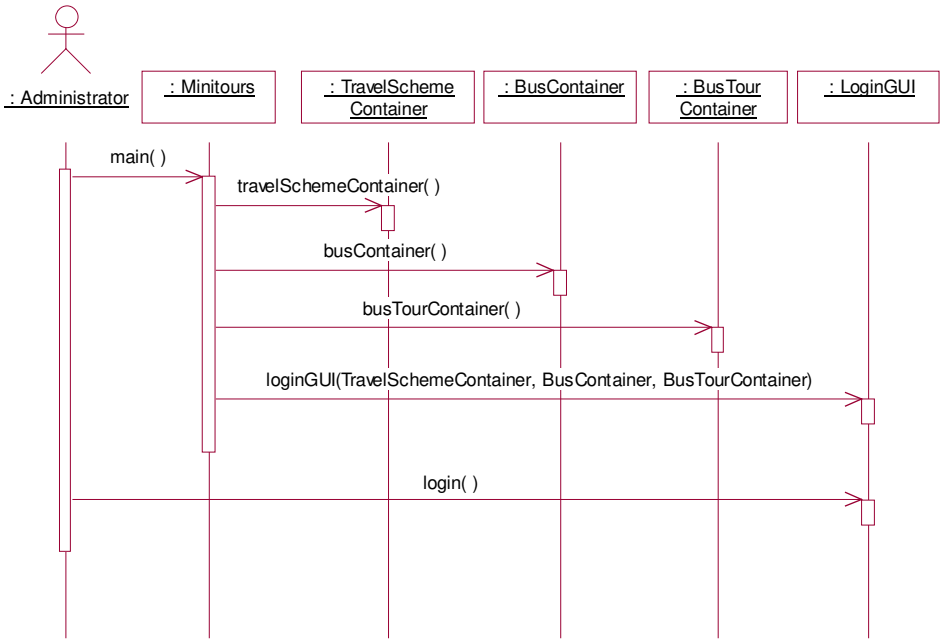


Abbildung 138: Sequenzdiagramm zum Systemstart

Beginnend mit dem Sequenzdiagramm aus der Abbildung 138 wird das Interface der Klasse ‘Minitours’ mit der Methode main()’ erstellt. Dargestellt ist dieses in der Abbildung 139. Da es sich um eine statische Methode handelt, d.h. diese bei der Klasse aufgerufen wird und somit keine Objekte der Klasse Minitours benötigt werden, enthält das Interface keine Signatur für einen Konstruktor. Durch das ‘C’ in der Kantenbeschriftung ‘(call,[C(class),M“main“]) ’ wird festgelegt, daß es sich um eine Nachricht an die Klasse und nicht an eine Ausprägung der Klasse handelt. Ein solches Empfängerobjekt ist mit ‘I’, für ‘instace’, statt mit ‘C’ gekennzeichnet. Die Kantenbeschriftung in der Abbildung 139 selektiert aus der Stelle inMsg alle Nachrichten an die Klasse Minitours mit dem Aufruf der Methode main(). Sofern die aufgerufene Methode Parameter enthält oder eine Selbstreferenz benötigt, werden diese Daten aus der Nachricht extrahiert und in den dafür bestimmten Stellen an die Methode übergeben. Die Methode ‘main()’ wird nur dadurch aufgerufen, daß eine Marke auf die mit ‘P out’ gekennzeichnete Kontrollstelle ‘main’ gelegt wird, d.h. diese Methode benötigt keine Parameter und keine Selbstreferenz.

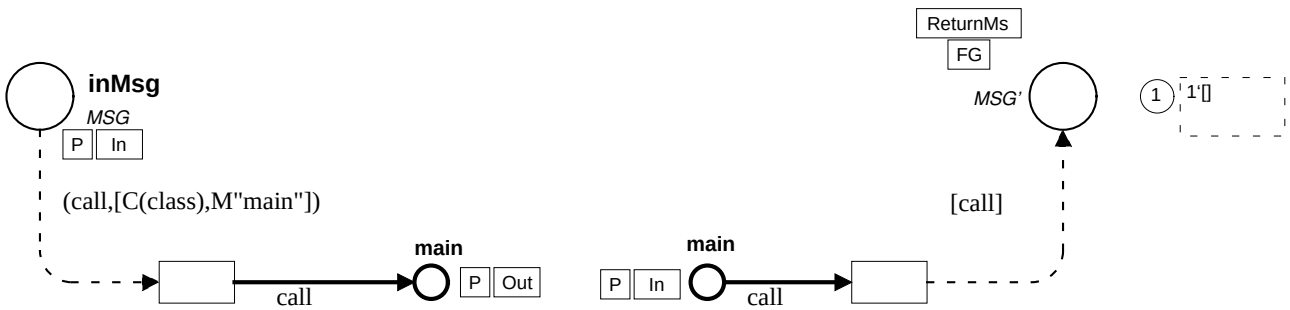


Abbildung 139: Interface der Klasse Minitours

In der Abbildung 140 ist die Klasse ‘Minitours’ dargestellt. Sie implementiert das Interface. Nachrichten werden durch das Interface auf der Unterseite ‘HS MinitoursInterface#74’ an die aufgerufene

Methode weitergereicht. Bei einer Nachricht an die Methode ‘main()’ wird eine Marke auf die Eingangsstelle ‘main’ der Methodentransition ‘main’ gelegt. Die Verfeinerung dieser Methode zeigt die Abbildung 141 und wird weiter unten erläutert. Die Marke auf der Stelle ‘Class’ auf der Klassen-Seite in der Abbildung 140 enthält den Namen der Klasse. Die Beschriftung der Transition ‘HS Class#5’ ist ein Beispiel für die Verwendung des Hierarchiekonzeptes mit der Verfeinerung von Transitionen auf Unterseiten. Die Class-Seite unterscheidet zwischen Nachrichten mit dem Aufruf der new-Methode und anderen Klassenmethoden bei Klassen und Nachrichten mit dem Aufruf von Methoden bei Ausprägungen von Klassen (Objekten). Um später die Simulation des Netzes zu starten, ist die Stelle ‘inMsg’ mit einer Marke mit dem Aufruf der Methode ‘main()’ initialisiert. Diese Marke ist als Initialmarkierung neben die Stelle inMsg geschrieben und hat die Form: (I((“Main“,~1)),[C(“Minitours“),M“main“]).

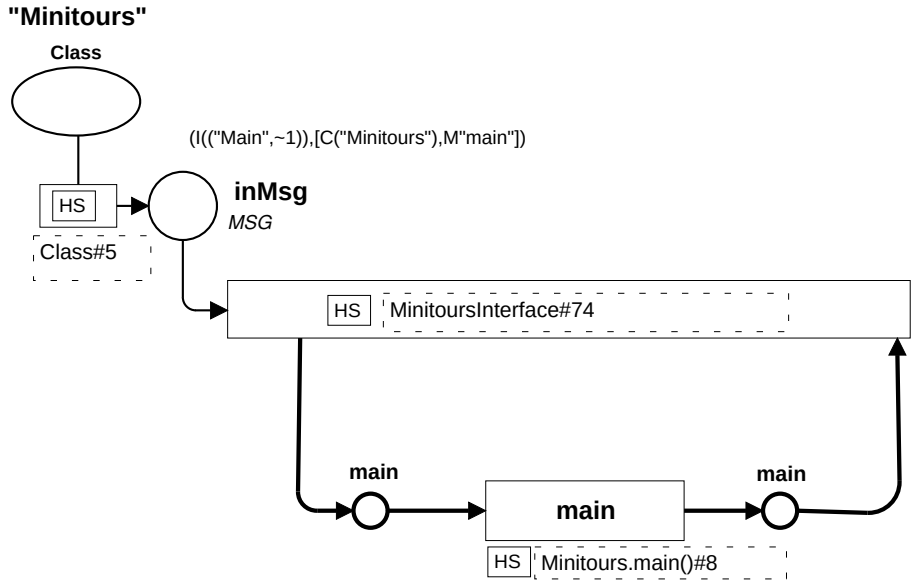


Abbildung 140: Klasse Minitours

Wird die Transition der main-Methode durch die Markierung der Eingangsstelle ‘main’ aktiviert, so schaltet zunächst die fork-Transition in der Abbildung 141. Dadurch werden die drei Kontrollstellen, die Ausgangsstellen dieser Transition sind, markiert. Dies bedeutet die Aktivierung von drei Transitionen, die jeweils eine Nachricht generieren. Eine Nachricht ruft die new-Methode der Klasse ‘TravelSchemeContainer’, eine die new-Methode der Klasse ‘BusContainer’ und eine die new-Methode der Klasse ‘BusTourContainer’ auf. Die Antwortnachrichten enthalten die Referenzen auf die erzeugten Objekte als Parameter und diese werden in den Stellen ‘TravelSchemeContainer’, ‘BusContainer’ und ‘BusTourContainer’ abgelegt. Die join-Transition führt den Kontrollfluß wieder zusammen. Zu diesem Zeitpunkt ist sichergestellt, daß die Erzeugung aller benötigten Containerobjekte erfolgreich abgeschlossen wurde. Dies ist wichtig, da diese Erzeugung in beliebiger Reihenfolge, d.h. nebenläufig, ausgeführt werden kann, während die im nächsten Schritt aufgerufene new-Methode der Klasse ‘LoginGUI’ erst nach der Erzeugung der Container aufgerufen werden darf. Die explizite Modellierung des Kontrollflusses ermöglicht eine gute Kontrolle über die Funktionalität der Netze. Modellierungsfehler können aufgrund von liegengebliebenen oder duplizierten Kontrollmarken gefunden und beseitigt werden. Der Aufruf der new-Methode der Klasse ‘LoginGUI’ erfolgt analog zu dem Aufruf der new-Methoden der Container, d.h. es wird eine Nachricht der Form ‘(call,[C(“LoginGUI“),M“new“]) ’ an die Klasse ‘LoginGUI’ geschickt. Die Antwortnachricht enthält die Referenz auf das neu erzeugte Objekt und diese wird von der nächsten Transition verwendet, um eine Nachricht mit dem Aufruf der Methode ‘login()’ an dieses Objekt zu senden. Alle Abläufe

im System gehen dann von dem Anmelden einer Person aus. Wenn das System heruntergefahren wird, endet die Abarbeitung der Methode ‘main()’ und die durch ‘P Out’ gekennzeichnete Stelle ‘main’ ist markiert.

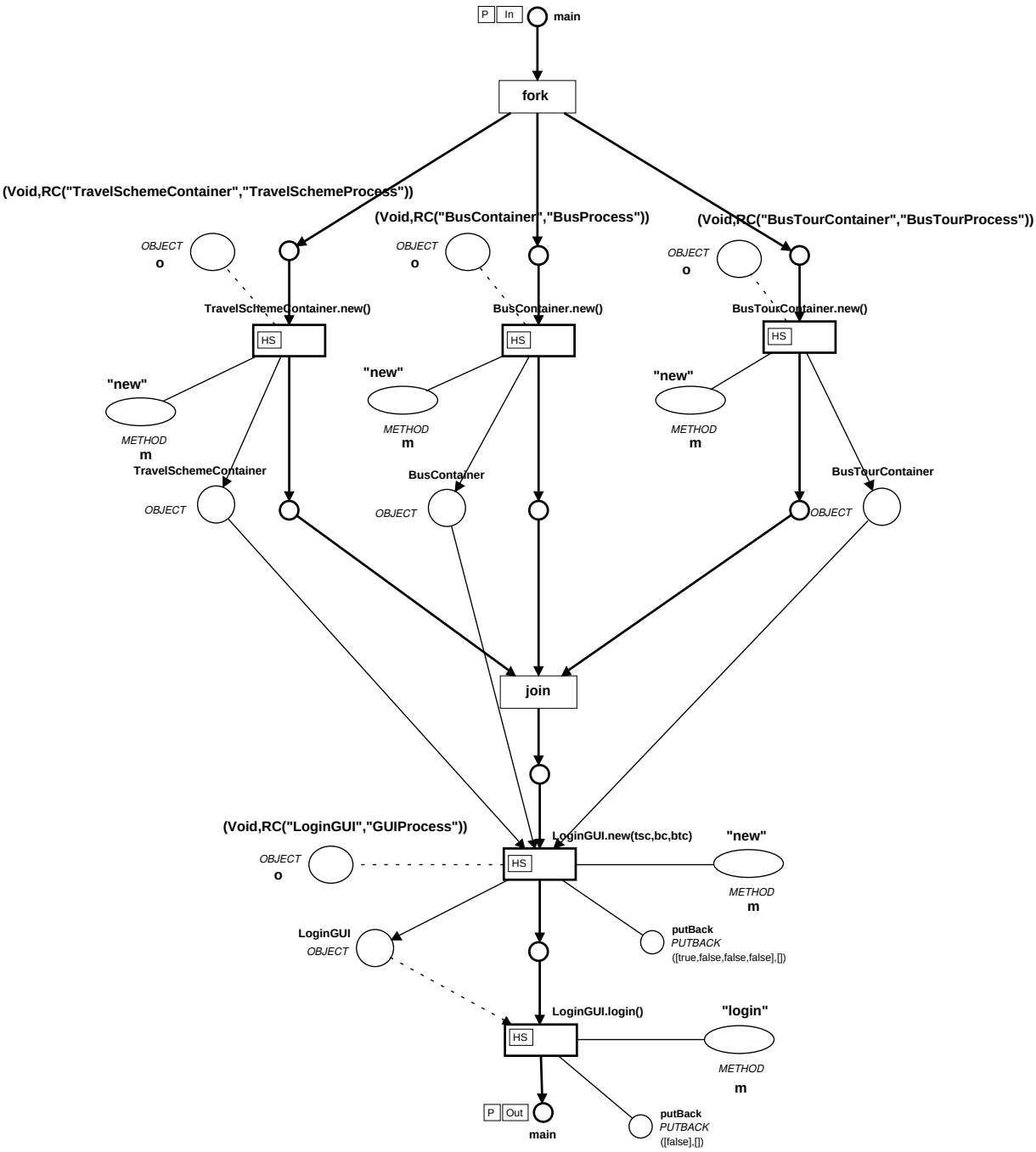


Abbildung 141: Verfeinerung der Methode main()

Stellvertretend für die Aufrufe der new-Methoden bei den Containerklassen wird hier die new-Methode der Klasse ‘TravelSchemeContainer’ betrachtet. Diese ist in der Abbildung 142 dargestellt. Das Interface und die das Interface implementierende Klasse werden in den oben aufgeführten Abbildungen in Kapitel 5.1 gezeigt. Die new-Methode benötigt die Selbstreferenz, um diese in einer Stelle abzulegen, die Referenzen auf die existierenden TravelSchemeContainer speichert und um diese als Rückgabewert auf die Ausgangsstelle_new zu legen. Außerdem wird bei der new-Methode die Zäh-

lerstelle für die aggregierten TravelSchemes mit dem Integerwert ‘0’ initialisiert.

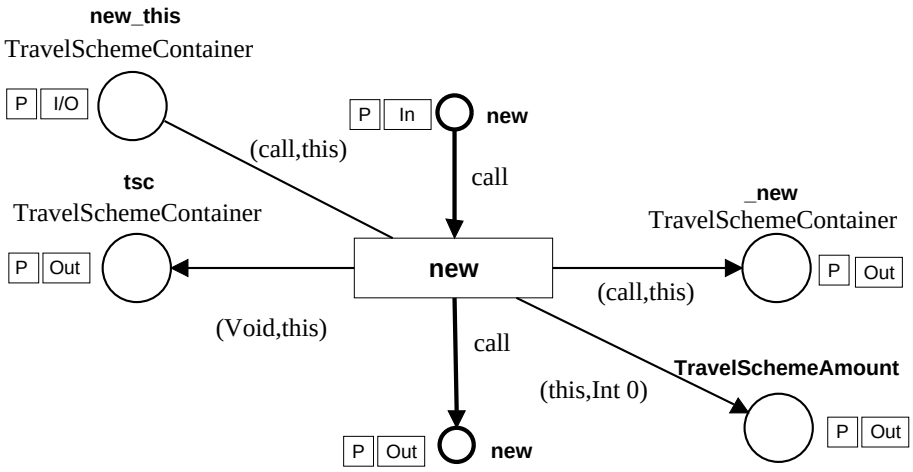


Abbildung 142: new-Methode der Klasse TravelSchemeContainer

Die new-Methode der Klasse ‘LoginGUI’ erhält die Referenzen auf die drei erzeugten Container als Parameter und speichert diese in lokalen Stellen. Damit kann das erzeugte LoginGUI-Objekt diese Referenzen bei dem Aufruf der new-Methode der Klasse ‘ManagerGUI’ als Parameter übergeben. In der Abbildung 143 ist die Methode ‘login()’ der Klasse ‘LoginGUI’ abgebildet. Durch die Code-Region der ersten durch die Kontrollmarke aktivierten Transition wird der Benutzer zu einer Eingabe aufgefordert. Tippt dieser ‘Manager’ ein, so erfolgt der Aufruf der new-Methode der Klasse ‘ManagerGUI’. Lautet die Eingabe ‘Client’, so wird die new-Methode der Klasse ‘ClientGUI’ aufgerufen. Die für den new-Aufruf benötigten Parameter wurden bereits bei der Erzeugung des LoginGUI-Objekts als Marken in die Stellen ‘tsc’, ‘bc’ und ‘btc’ gelegt.

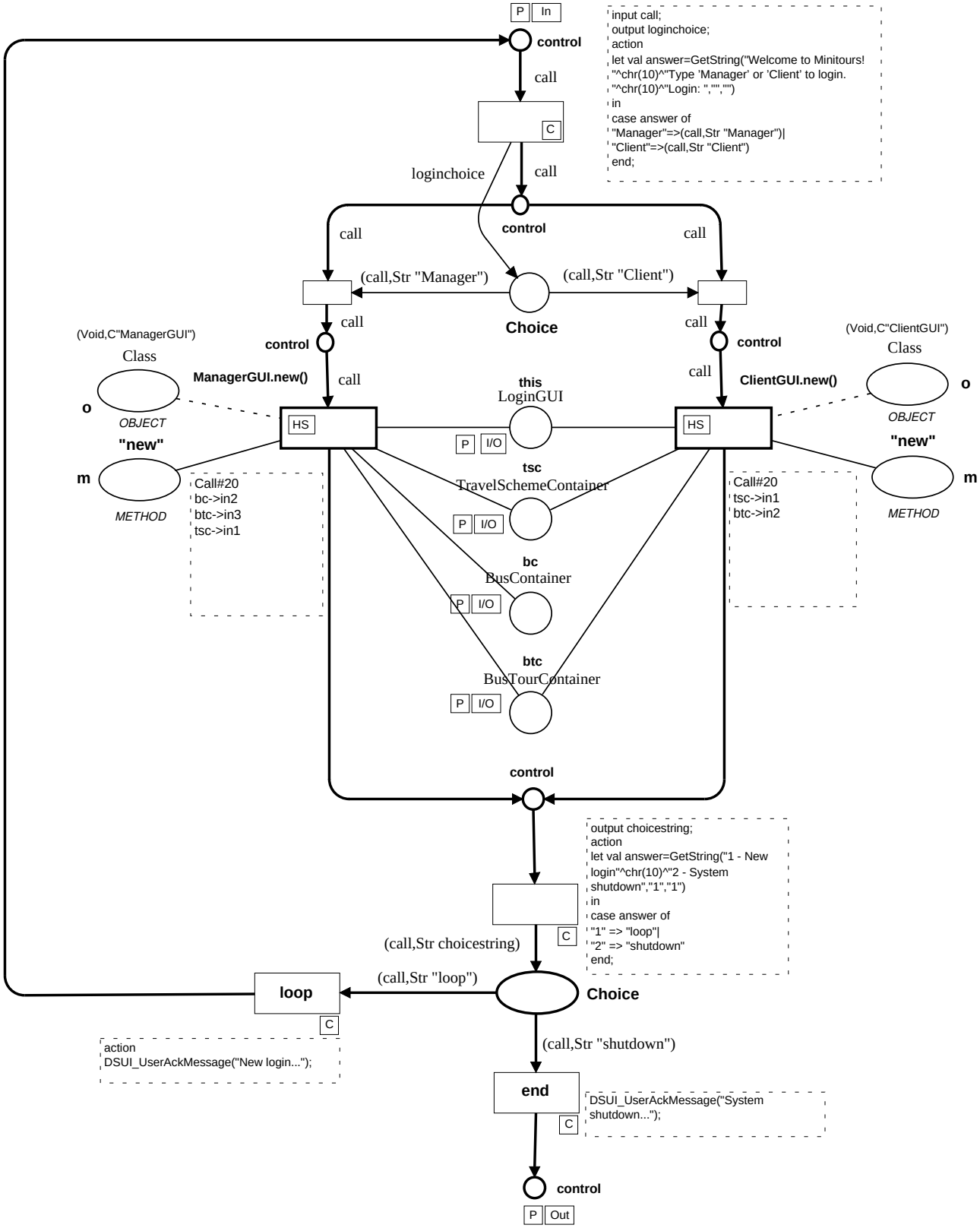


Abbildung 143: Methode login()

Im folgenden wird das Anmelden eines Managers nachvollzogen. Auf die Darstellung des Interface und der Klasse wird hier verzichtet und auf den Anhang verwiesen. Die new-Methode der Klasse ‘ManagerGUI’ zeigt die Abbildung 144. Die Referenzen auf die Container werden lokal in den Stellen ‘tsc’, ‘bc’ und ‘btc’ gespeichert, um Nachrichten an diese Container schicken zu können. Als

nächstes erfolgt der Aufruf der Methode ‘menu()’, aus der Abbildung 145, bei dem ManagerGUI selbst. Diese bietet dem Benutzer ein Menü an, aus dem ein Menupunkt ausgewählt werden kann. Je nachdem für welchen Menüpunkt sich der Benutzer entscheidet, wird eine weitere Methode des ManagerGUIs aufgerufen.

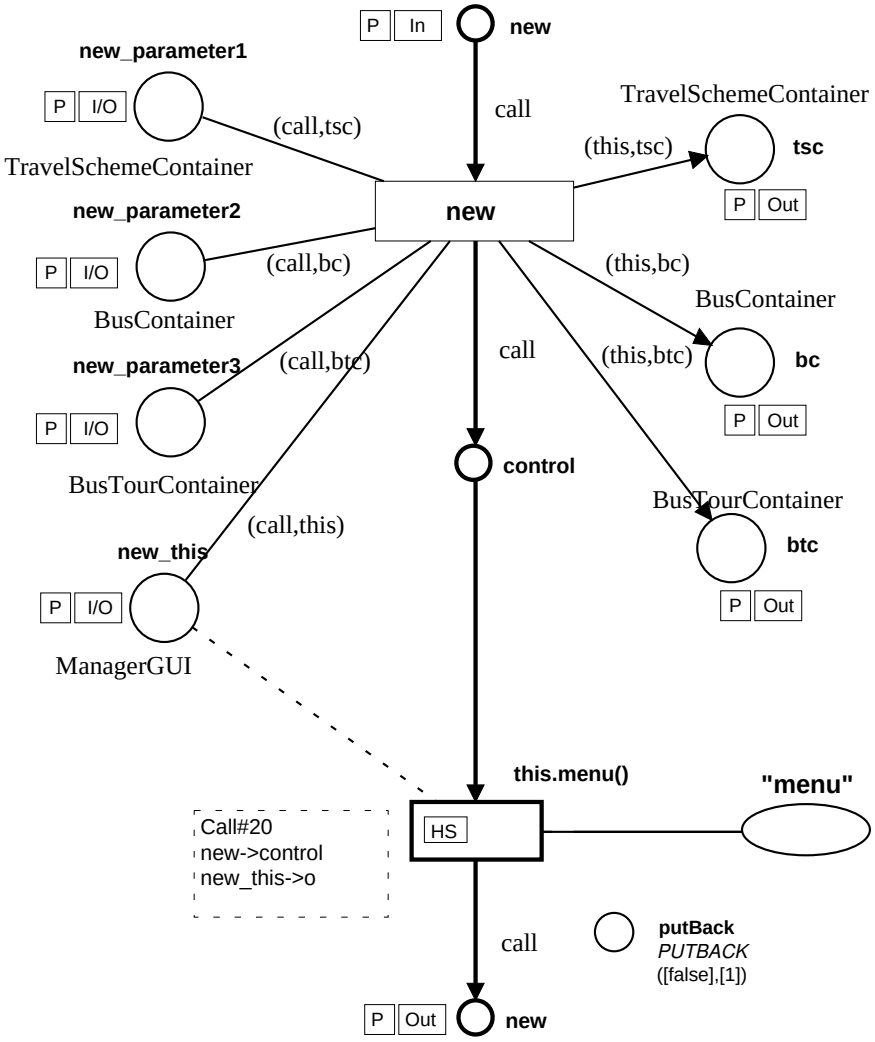


Abbildung 144: new-Methode der Klasse ManagerGUI

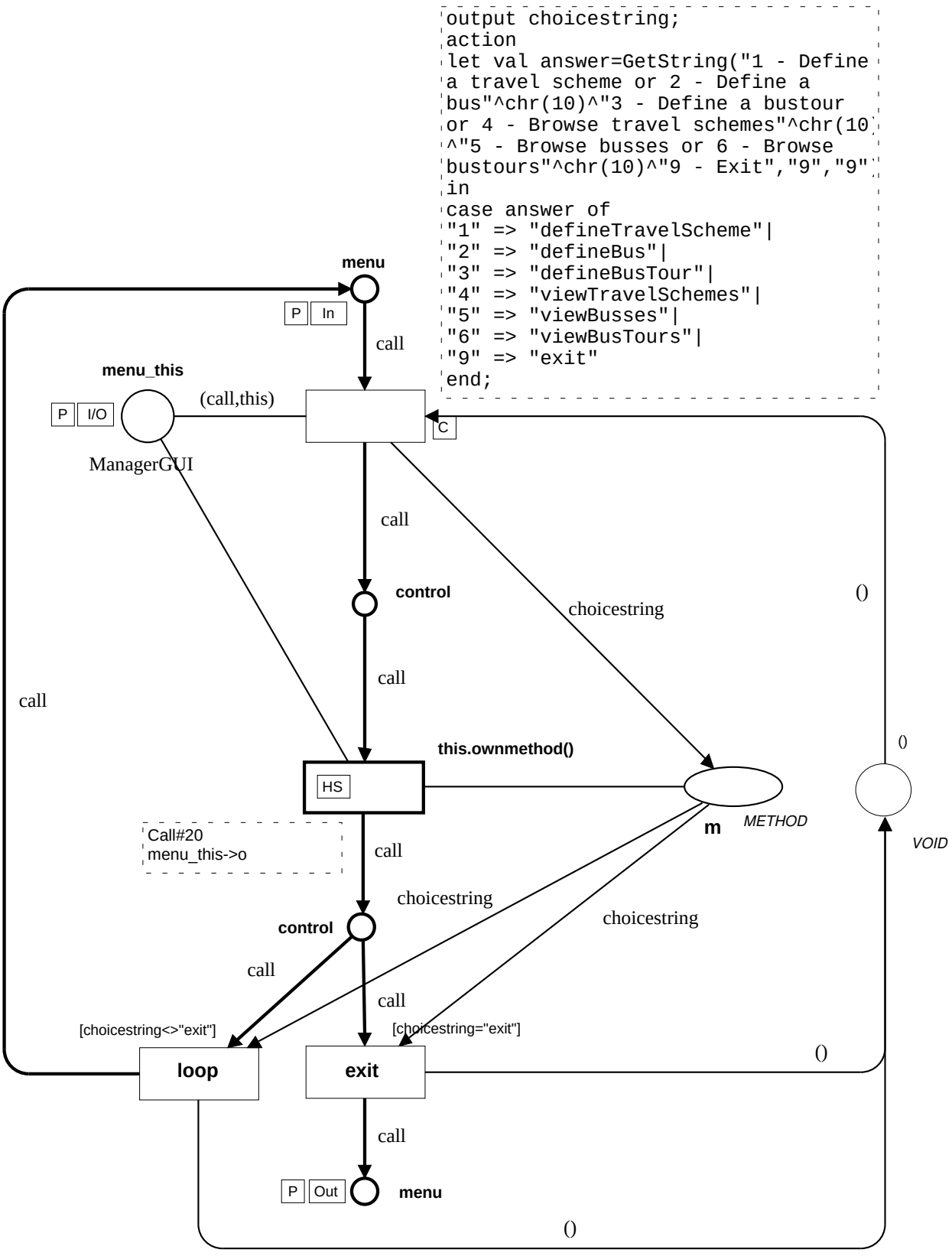


Abbildung 145: Methode menu()

Bei der Auswahl des ersten Menüpunktes (defineTravelScheme) erfolgt der Aufruf der Methode ‘createTravelScheme()’ des Objekts vom Typ ‘TravelSchemeContainer’, dessen Referenz in der Stelle ‘tsc gespeichert’ wurde. Die Abbildung 146 zeigt die Methode ‘defineTravelScheme()’, die

diesen Aufruf tätigt. Die Methode ‘createTravelScheme()’ ist weiter oben abgebildet (Abbildung 135).

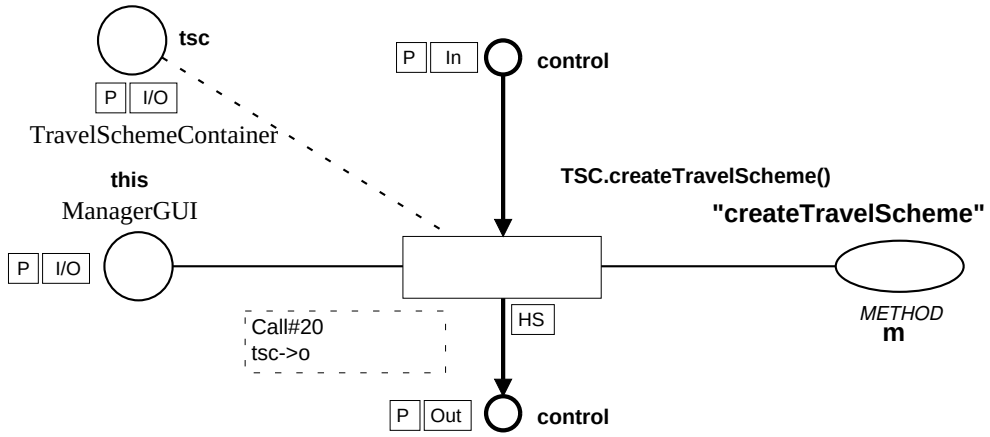


Abbildung 146: Methode defineTravelScheme

Bei der Abarbeitung der Methode ‘createTravelScheme()’ wird zuerst die new-Methode der Klasse ‘TravelScheme’ aufgerufen. Der Rückgabewert ist die Referenz auf das erzeugte Objekt vom Typ TravelScheme und wird an die Methode ‘storeTravelScheme()’ als Parameter übergeben, welche nach der Beendigung der new-Methode aufgerufen wird. Durch diesen Vorgang ist die Referenz auf das neuerzeugte Objekt vom Typ ‘TravelScheme’ bei dem aggregierenden Objekt vom Typ ‘TravelSchemeContainer’ gespeichert. In der Abbildung 147 ist die erwähnte new-Methode der Klasse ‘TravelScheme’ dargestellt. Wird diese aufgerufen, so werden nach der Erzeugung einer neuen Objekt-ID zuerst die als Stellen modellierten Instanzvariablen (Attribute) des neuen Objekts mit Marken, die zunächst keine Werte enthalten, initialisiert. Nach dieser Initialisierung schaltet die fork-Transition und aktiviert sieben Transitionen, die nebenläufig Nachrichten generieren, um den Attributen Werte zuzuweisen. Die aufgerufenen Methoden befinden sich auf der TravelScheme-Klassen-Seite und sind nicht auf Unterseiten dargestellt, weil die Benutzerabfrage sehr kompakt in Code-Regionen implementiert werden kann und hier einige Unterseiten vermieden werden konnten. Nachdem alle Werte gesetzt sind, wird der Kontrollfluß durch die join-Transition wieder zusammengeführt. Die letzte Transition der new-Methode legt den Rückgabewert der Methode auf die Stelle ‘_new’.

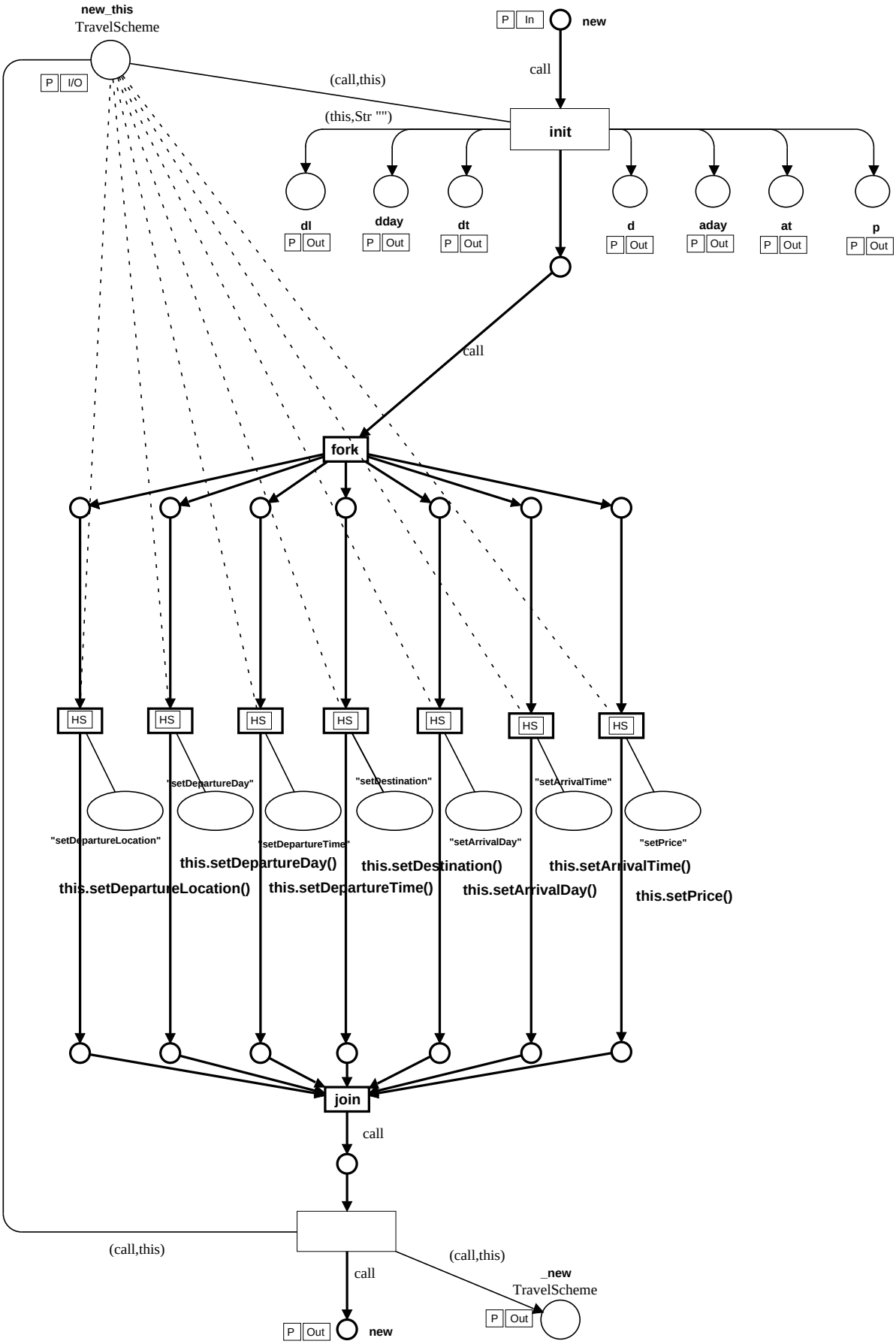


Abbildung 147: new-Methode der Klasse TravelScheme

Nach der Ausführung der Methoden, die zur Definition eines Reiseschemas nötig sind, befindet sich die Kontrollmarke in der Kontrollstelle, die Ausgangsstelle der Transition ‘this.ownmethod()’ in der Methode ‘menu()’ des ManagerGUIs ist (Abbildung 145). Da es sich bei dem gewählten Menüpunkt nicht um Nummer 9 (exit) handelte, ist die Transition ‘loop’ mit dem Guard ‘[choicestring<>“exit“] ’ aktiviert und legt beim Schalten die Kontrollmarke in die durch ‘P In’ gekennzeichnete Stelle. Außerdem wird die zu Beginn aus der Stelle ‘VOID’ entnommene Marke zurückgelegt. Sie verhindert den nebenläufigen Aufruf der Methode ‘menu()’, da dies nicht erwünscht ist. Wählt der Benutzer den Menupunkt Nummer 9 (exit), so schaltet die Transition mit dem Guard ‘[choicestring=“exit“]’ und die Methode ‘menu()’ ist beendet. Damit gelangt die Kontrollmarke in die mit ‘P Out’ gekennzeichnete Kontrollstelle ‘new’ in der Methode ‘new’ des ManagerGUIs. Durch das Beenden dieser new-Methode wird die Kontrollmarke in die Ausgangsstelle ‘control’ der Transition ‘GUI.new()’ in der Methode ‘login()’ des LoginGUIs (Abbildung 143) gelegt. Somit ist die unbenannte Transition mit der Code-Region aktiviert und verlangt beim Schalten eine Benutzereingabe.

Wünscht der Benutzer eine neue Anmeldung, so gibt er eine ‘1’ ein und damit ist die Zeichenkette ‘loop’ in der erzeugten Marke enthalten und die Transition loop ist aktiviert. Beim Schalten erfolgt eine informative Ausgabe an den Benutzer und die Kontrollmarke wird in die Startstelle der Methode ‘login()’ gelegt. Wählt der Benutzer das Herunterfahren des Systems, so wird die Transition ‘end’ aktiviert. Wenn diese schaltet wird, neben einer Ausgabe an den Benutzer, die Kontrollmarke in die mit ‘P Out’ beschrifteten Stelle ‘main’ in der Methode ‘main’ (Abbildung 141) gelegt. Die Methode ‘main()’ ist damit beendet und es gibt nach dem Schalten der rechten Transition auf der Minitours Interface-Seite (Abbildung 139) keine weiteren aktivierten Transitionen.

Die hier vorgestellten Klassen wurden in dieser Vorgehensweise schrittweise erzeugt. D.h. zuerst wurde ein Interface für eine Klasse generiert, dann die dieses Interface implementierende Klassen-Seite und dann die Methoden. Diese wurden entsprechend den Szenarien schrittweise implementiert und das Petrinetz war zu jedem Zeitpunkt ein ausführbarer Prototyp. Die Transitionen der noch nicht implementierten Methoden schalten einfach die Marken durch und können zu Testzwecken mit Ausgabefunktionen in den Code-Regionen versehen werden. Werden bei der Netzmodellierung Fehler erkannt, so werden diese im Klassendiagramm, den Interaktionsdiagrammen und Zustandsdiagrammen korrigiert. Dies kann weitere Änderungen in den Diagrammen verlangen, welche sich wiederum auf die Petrinetze auswirken. Durch dieses iterative Vorgehen werden alle Entwicklungsphasen optimal angewendet.

5.3 Modellierungsentscheidungen

In diesem Abschnitt werden einige Modellierungsentscheidungen diskutiert. Bei dieser Diskussion geht es nicht darum, das Modell des Reiseunternehmens zu diskutieren, sondern die Umsetzung des UML-Modells in objektorientierte Petrinetze. Dies betrifft die Klassen mit den Attributen und Methoden und die Assoziationen zwischen den Klassen, inklusive Aggregation und Vererbung. Betrachtet werden außerdem Nebenläufigkeitsaspekte bezüglich der Implementation von Methoden.

Als Diskussionsgegenstand werden in diesem Abschnitt die bereits bekannten Klassen ‘TravelSchemeContainer’ und ‘TravelScheme’, sowie die Klassen ‘TravelData’, ‘BusTourContainer’ und ‘BusTour’ betrachtet. Zwischen ‘TravelSchemeContainer’ und ‘TravelScheme’ besteht eine Aggregationsbeziehung, ‘TravelScheme’ erbt von der abstrakten Klasse ‘TravelData’ und es besteht eine Kann-Beziehung von der Klasse ‘TravelScheme’ zur Klasse ‘BusTour’ und eine Muß-Beziehung in die andere Richtung. Die Abbildung 148 zeigt die erwähnten Klassen im Klassendiagramm. Zuerst erfolgt die Beschreibung der Attribute der Klassen. Die Klasse ‘TravelScheme’ und ‘BusTour’ haben beide die von der Klasse ‘TravelData’ geerbten Attribute ‘departureLocation’, ‘departureTime’, ‘de-

stination', 'arrivalTime' und 'price'. Zusätzlich hat die Klasse 'TravelScheme' die Attribute 'departureDay' und 'arrivalDay', um Wochentage (Mo - So) als Werte für ein Reiseschema festzulegen. Die Klasse 'BusTour' hat die zusätzlichen Attribute 'departureDate' und 'arrivalDate', um ein konkretes Datum (z.B. 29.01.1999) für eine geplante Busreise zu speichern. Die Containerklassen haben keine expliziten Attribute. Weiter unten wird aber gezeigt, daß diese Klassen Referenzen auf ihre aggregierten Objekte verwalten und somit lokal verfügbar haben müssen. Da diese Referenzspeicher technischer Art für Container sind und durch die Art der Assoziation (Aggregation) definiert sind, werden sie hier nicht als explizite Attribute betrachtet. Ein Attribut anderer Art eines Containers könnte z.B. eine Angabe über dessen Kapazität sein. Diese wird hier als beliebig groß angenommen. Die Attribute werden in den Petrinetzen als Stellen modelliert. Eine Marke in einer solchen Attribut- oder auch Instanzvariablenstelle besteht immer aus dem Wert des Attributs und der Referenz auf das Objekt, um dessen Attribut es sich handelt. Dies ist notwendig, weil bei den hier verwendeten Petrinetzen mehrere Objektnetze der gleichen Klasse zusammengefaltet werden, so daß sich die Marken des gleichen Attributs unterschiedlicher Objekte alle in derselben Stelle befinden. Eine Marke des Attributs 'price' mit dem Wert '120,- ' für das Objekt mit der Referenz '("TravelScheme",1)' hat dann die Form '(("TravelScheme",1),Str "120,-")'. Eine weitere Marke für ein anderes Objekt mit der Referenz '("TravelScheme",2)' und dem Wert '90,- ' für das Attribut 'price' hat die Form '(("TravelScheme",2),Str "90,-")'.

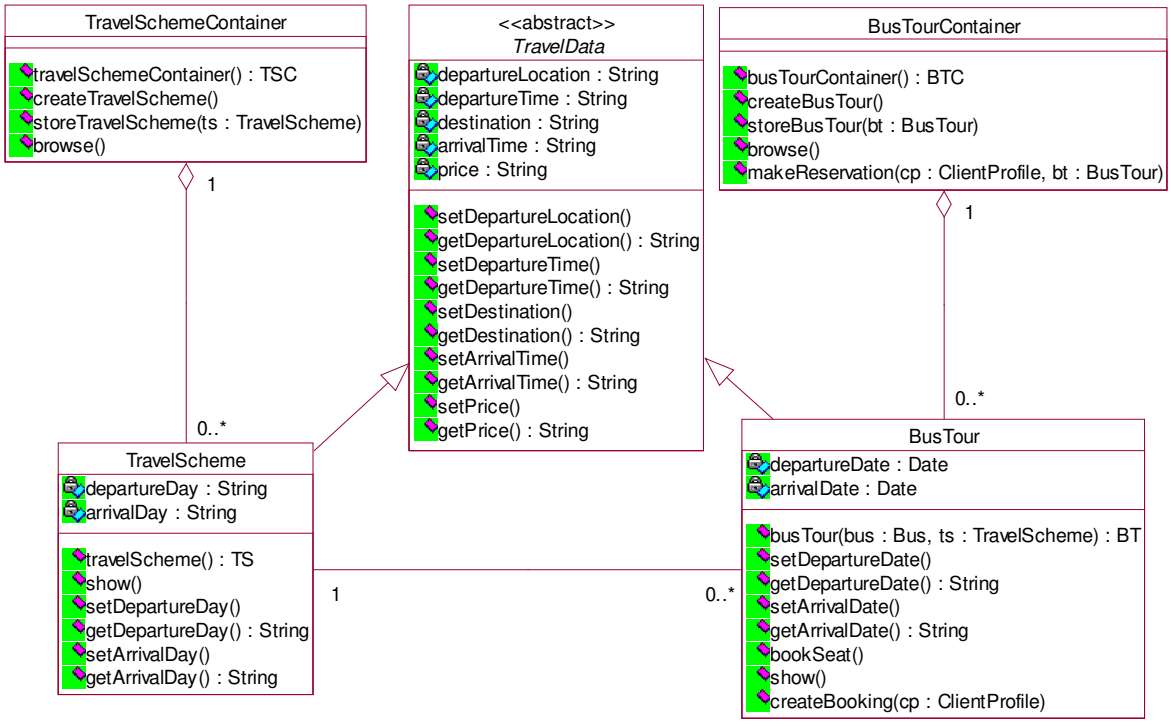


Abbildung 148: Klassendiagramm

Bei der Klasse 'TravelData' handelt es sich um eine abstrakte Klasse, so daß es keine Ausprägungen dieser Klasse geben kann. Die Methoden werden erst in den Klassen implementiert, die von dieser Klasse erben. In diesem Fall sind dies die Klassen 'TravelScheme' und 'BusTour'. Deren Konstruktormethoden fordern den Benutzer zum Setzen der Attributwerte durch Aufruf der set-Methoden auf. Die new-Methode der Klasse 'TravelScheme' wurde bereits weiter oben in der Abbildung 147 dargestellt. Der Aufruf der set-Methoden erfolgt nebenläufig. Mit den Petrinetzen könne solche ne-

benläufigen Aktivitäten sehr gut modelliert werden. Bei der Spezifikation von Methoden brauchen keine Abläufe künstlich sequentialisiert zu werden. Analog werden die get-Methoden in der Methode ‘show()’ nebenläufig aufgerufen. Nachdem alle Werte ermittelt wurden, werden diese durch die Verwendung einer Code-Region angezeigt (Abbildung 149). Die set-Methoden speichern die eingegebenen Werte in der oben vorgestellten Form in Stellen für die Attribute. Die Kantenbeschriftung lautet deshalb ‘(this,Str instring) ’, wobei ‘this’ die Referenz des gerade aktiven Objekts ist und ‘instring’ der vom Benutzer eingegebene Wert.

Methoden eines Objekts können mehrfach zu sich selbst nebenläufig aufgerufen werden, sofern dies nicht explizit in der Modellierung ausgeschlossen wird. Diese nebenläufigen Aufrufe unterscheiden sich durch die Aufrufidentifikation ‘call’. Zum Beispiel würden bei einem mehrfachen, nebenläufigen Aufruf einer get-Methode eines Objekts die ermittelten Werte an die durch ‘call’ unterscheidbaren Aufrufe zurückgeschickt. Zur Illustration dient hier die Abbildung 150 mit einem Ausschnitt aus der Klasse ‘TravelScheme’. Eine set-Methode eines Objekts soll nicht nebenläufig zu sich selbst schalten können, weil dies die bekannten Probleme beim gleichzeitigen Schreiben mehrerer Schreiber verursachen würde. Die Nebenläufigkeit wird dadurch eingeschränkt, daß die den Wert enthaltende Marke aus der Attributstelle herausgenommen wird, bevor die Marke mit dem neuen Wert hineingelegt wird. Während ein Schreiber die Marke besitzt, ist keine andere Transition, die diese Marke in einer Eingangsstelle zum Schalten benötigt, aktiviert. Also im hier betrachteten Beispiel weder eine set- noch eine get-Methode.

Eine andere Möglichkeit, das nebenläufige Schalten einer Transition zu sich selbst einzuschränken, besteht in der expliziten Modellierung einer Stelle, die die Schaltbedingung als Marke(n) enthält. Im einfachsten Fall handelt es sich um eine Marke vom Typ ‘Void’, d.h. eine Marke, die keine unterschiedlichen Werte annehmen kann. Die Methode, die diese Marke zum Schalten als Nebenbedingung benötigt, kann dann nicht nebenläufig zu sich selbst ausgeführt werden. Im Fallbeispiel findet man eine solche Marke bei der Modellierung der menu()-Methoden. Durch das Erhöhen der Markenanzahl kann der Grad der Nebenläufigkeit kontrolliert werden. Soll eine Methode nur einmal ausgeführt werden dürfen, so wird die Marke nicht in die Stelle zurückgelegt. Die Methode kann nie wieder aufgerufen werden, es sei denn, daß durch eine andere Methode eine Marke in die Stelle gelegt wird, wenn diese Funktionalität gewünscht ist. Die Methode ‘register()’ in der Klasse ‘ClientGUI’ ist auf diese Weise modelliert worden. Ein Kunde kann sich somit nur einmal registrieren.

Der wechselseitige Ausschluß der store- und browse-Methoden in den Container-Klassen wird über die Marke mit der Information über die Anzahl der aggregierten Objekte realisiert. Sowohl die store- als auch die browse-Methoden ziehen diese Marke während der Ausführung von der Stelle ab und legen sie erst nach der vollständigen Abarbeitung der Methode wieder zurück. Die Abbildung 134 weiter vorn in diesem Kapitel zeigt diese Modellierung. Dieser wechselseitige Ausschluß könnte auch über eine explizit modellierte Stelle mit einer Marke vom Typ ‘Void’ realisiert werden.

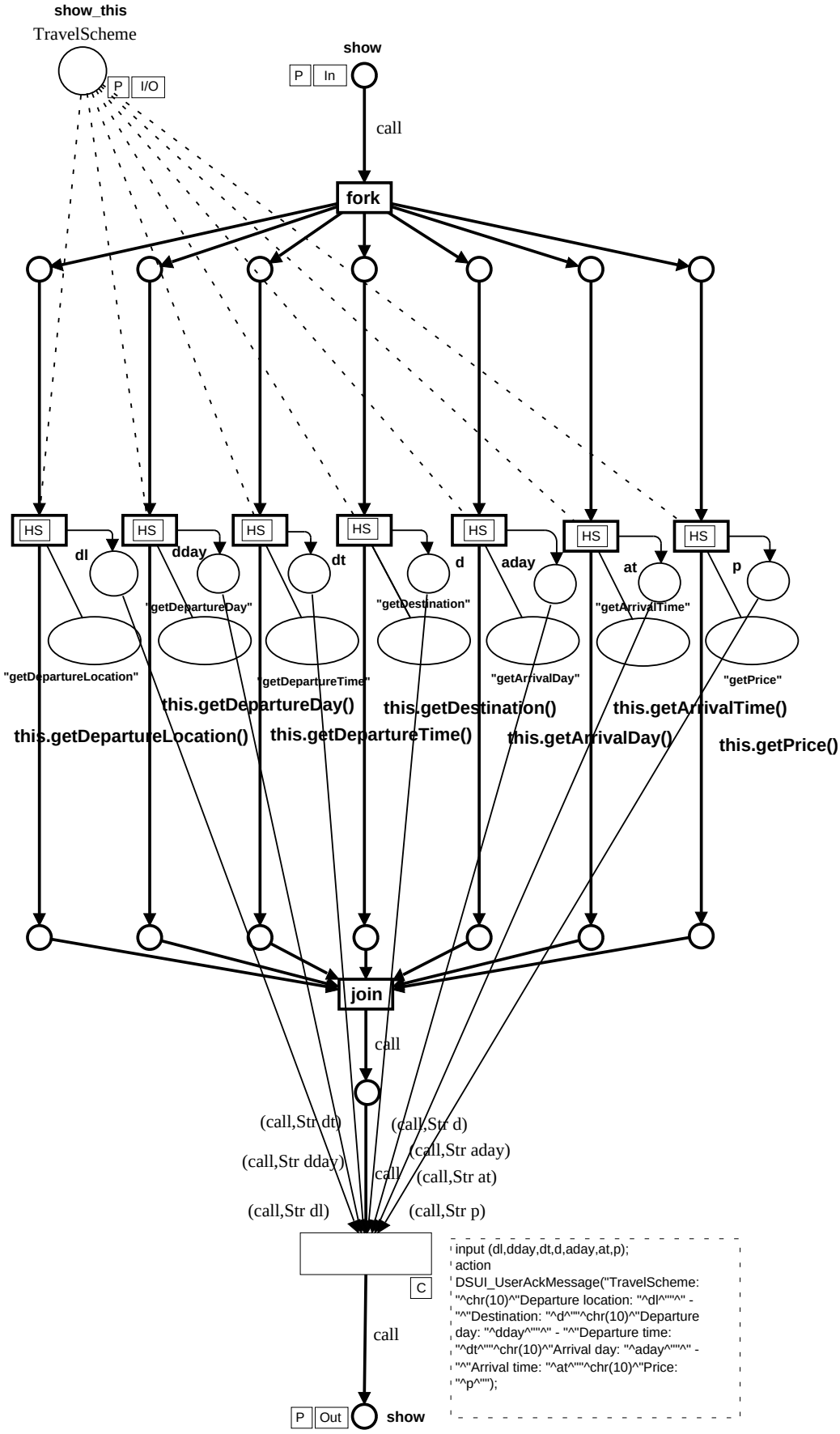


Abbildung 149: Methode show() der Klasse TravelScheme

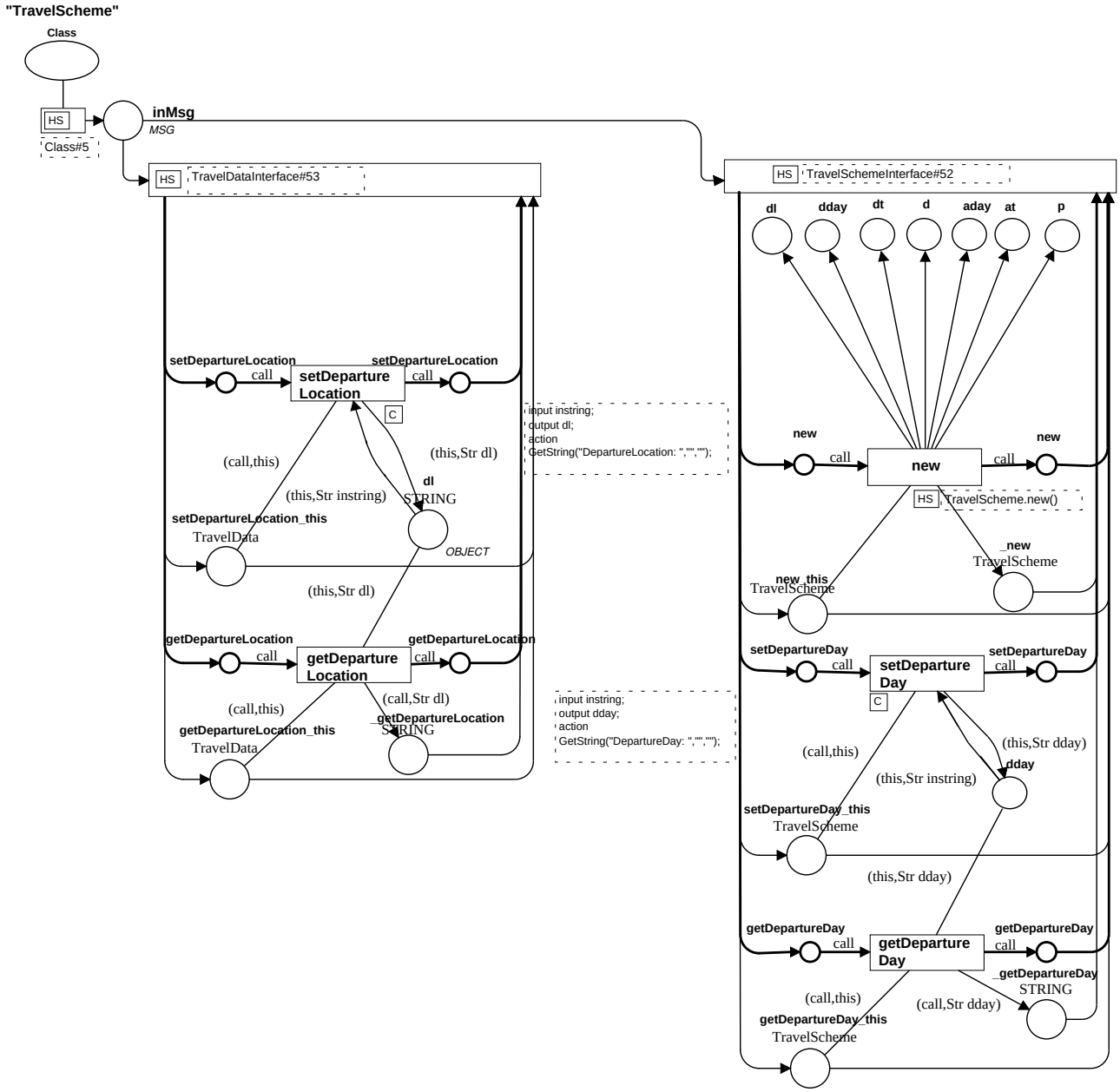


Abbildung 150: set- und get-Methoden

Die Aggregationsbeziehung wird durch Stellen in dem aggregierenden Objekt modelliert, die Referenzen auf die aggregierten Objekte enthalten. Ein Objekt vom Typ ‘BusTourContainer’ enthält eine Stelle ‘BusTours’, in der die Referenzen auf die aggregierten Busreisen gespeichert werden. Die Abbildung 151 zeigt diese Stelle in der Klasse ‘BusTourContainer’. Eine Marke in dieser Stelle enthält die Information über die Identifikation des Objekts vom Typ ‘BusTourContainer’ und die zu diesem gespeicherte Referenz auf eine Busreise. Die aggregierten Objekte haben keine Referenz auf den Container, von dem sie aggregiert werden.

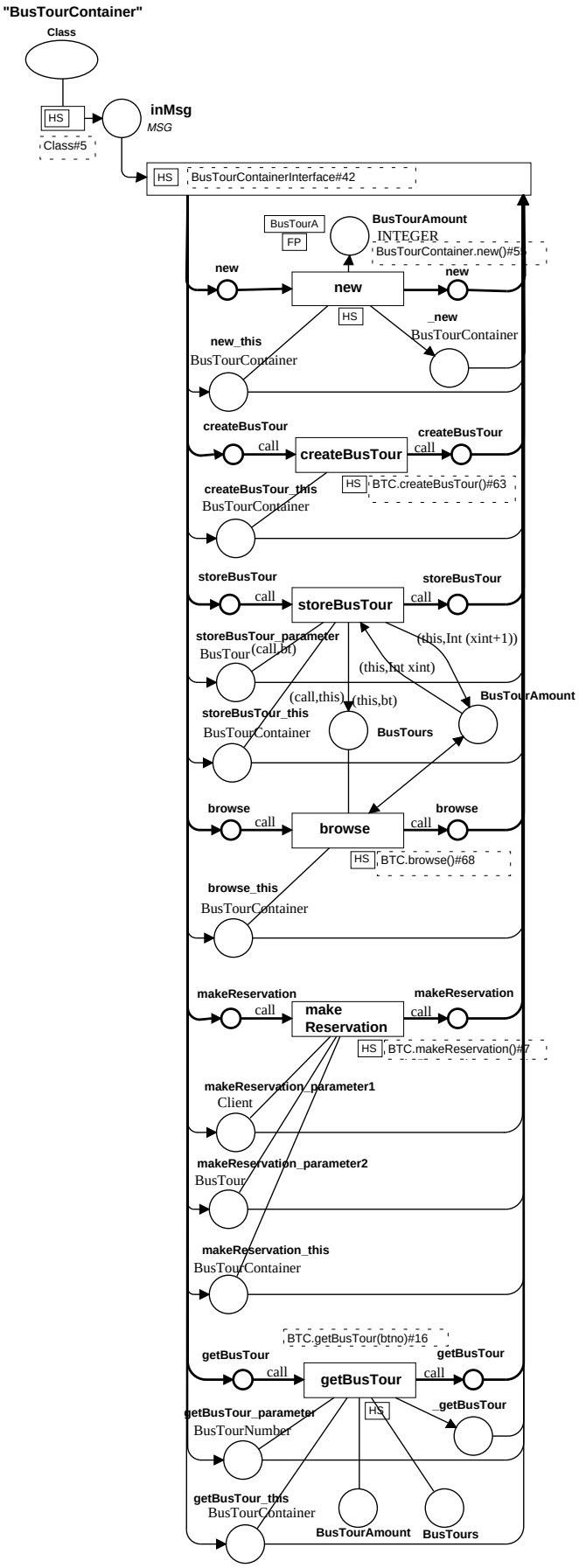


Abbildung 151: Klasse BusTourContainer

Die Vererbungsbeziehung wird durch die Implementation mehrerer Interfaces durch eine Klasse modelliert. Die Klasse ‘TravelScheme’ aus der Abbildung 150 implementiert die Interfaces ‘TravelScheme’ und ‘TravelData’. Das Interface ‘TravelData’ wird von den beiden Klassen ‘TravelScheme’ und ‘BusTour’ implementiert. Da diese als abstrakte Klasse definiert wurde und keine Methodenimplementationen enthält, werden diese in die Klassen verlagert. Da in dem hier gezeigten Fall die recht einfachen, geerbten set- und get-Methoden auf den Klassen-Seiten direkt implementiert wurden und nicht auf Unterseiten, wird hier kein Code wiederverwendet. Bei der Modellierung der Methoden als Unterseiten können diese beliebig oft verwendet werden. Ein Beispiel für Code-Wiederverwendung sind die vom ManagerGUI und vom ClientGUI gemeinsam verwendeten Methoden ‘viewTravelSchemes()’ und ‘viewBusTours()’.

Die Assoziation zwischen der Klasse ‘TravelScheme’ und ‘BusTour’ bedeutet, daß es zu einer Busreise immer genau ein Reiseschema geben muß, es aber zu einem Reiseschema keine Busreise zu geben braucht. Dies impliziert, daß bei der Erzeugung einer Busreise überprüft werden muß, ob es ein korrespondierendes Reiseschema gibt. Außerdem muß bei dem Löschen eines Reiseschemas berücksichtigt werden, daß die zu diesem Schema definierten Busreisen mitgelöscht werden müssen, bzw. das Reiseschema nicht ohne weiteres gelöscht werden darf, weil die Busreisen nicht gelöscht werden dürfen.

Modelliert werden würde die Muß-Beziehung durch die Angabe einer Referenz auf ein Objekt der Klasse ‘TravelScheme’ bei der Erzeugung eines Objekts der Klasse ‘BusTour’. Damit wäre sichergestellt, daß es zu einer Busreise immer ein Reiseschema gibt. Der Fall, daß ein Reiseschema gelöscht werden soll, ist im System nicht vorgesehen. Wenn dies berücksichtigt werden würde, müßte ein Objekt vom Typ ‘TravelScheme’ Referenzen auf alle Busreisen, die eine Referenz auf dieses Reiseschema besitzen, haben, um diese zu löschen, bzw. um festzustellen, daß zu dem Reiseschema Busreisen definiert sind und dieses deswegen nicht gelöscht werden darf. Bei Kann-Beziehungen wird die Referenz bei der Erzeugung eines Objekts nicht benötigt.

Exemplarisch für die Realisierung einer Muß-Beziehung wird in dem hier vorgestellten Modell die Assoziation zwischen der Klasse ‘Bus’ und der Klasse ‘BusTour’ modelliert. Bei der Erzeugung eines Objekts der Klasse ‘BusTour’ muß eine Busnummer angegeben werden. Auf diesem Wege wird hier sichergestellt, daß es einen Bus zu einer Busreise gibt. Die Überprüfung der Existenz eines Busses mit der angegebenen Busnummer könnte leicht integriert werden. Zur korrekteren Modellierung würde die Referenz auf den entsprechenden Bus ermittelt und bei dem Busreise-Objekt gespeichert werden. Ein solcher Mechanismus wird bei der Ermittlung der zu buchenden Reise bei dem Aufruf der book–Methode bei einem Objekt der Klasse ‘ClientGUI’ realisiert. Aus Gründen der Komplexität wird er nicht für alle Muß-Beziehungen implementiert.

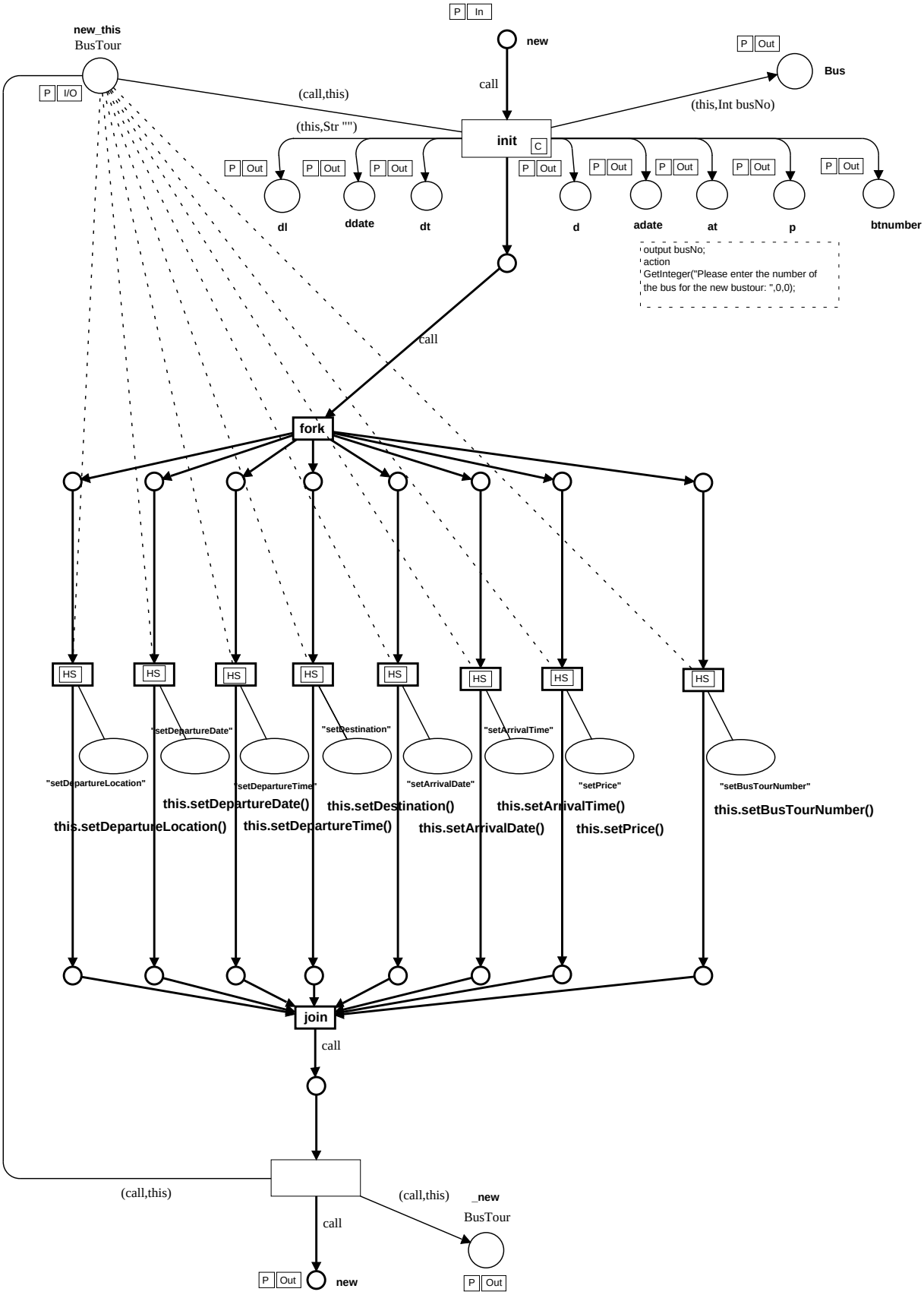


Abbildung 152: new-Methode der Klasse BusTour

In [Moldt 96] werden Assoziationen explizit durch Assoziationsklassen modelliert. Ausprägungen dieser Klassen haben spezielle Stellen, die Marken enthalten, in denen die Information über die Asso-

ziationen zwischen verschiedenen Objekten gespeichert ist. Auf diese Art können Muß- und Kann-Beziehungen modelliert werden, ohne die in den Beziehungen stehenden Objekte modifizieren zu müssen. Der Nachteil ist der Mehraufwand beim Erzeugen der Assoziationsklassen und dem Generieren der Marken für jede Ausprägung einer Assoziation.

Bei dem hier verwendeten Interface-Ansatz sind, wie oben erläutert, die Assoziationen nicht als eigene Klassen modelliert worden. Der Vorteil der Verlagerung der Assoziationen in die gewählten Klassen liegt in der geringeren Komplexität des Petrinetzes. Der Nachteil ist, daß die Klassen nicht ohne weiteres in einem anderen System wiederverwendet werden können, wenn sie dort in einer anderen Assoziation mit einer anderen Klasse stehen. Auch wenn sich nach der Implementation der Netze in diesem Modell Assoziationen ändern, müssen Klassen verändert werden, obwohl diese unabhängig von den Assoziationen in denen sie stehen, definiert sein sollten. Um effizient ein Klassendiagramm in Petrinetze zu überführen ist es aufgrund der Komplexität der Netze notwendig, ein Werkzeug zur Erzeugung der Interface-Seiten, Klassen-Seiten und der Assoziationsklassen zu haben. Dann kann der Implementation der Methoden die volle Zeit und Aufmerksamkeit gewidmet werden.

Die anderen Klassen des Minitours-Petrinetzmodells werden hier nicht diskutiert. Durch die grafische Repräsentation des Modells können alle Interface-, Klassen- und Methoden-Seiten im Anhang angesehen und die Szenarien nachvollzogen werden. Am besten eignet sich natürlich die Simulation des Modells zur Veranschaulichung des modellierten Systems. Bei der interaktiven Simulation kann das Schaltverhalten der Transitionen beobachtet werden. Dadurch lassen sich sehr gut die nebenläufigen bzw. sequentiellen Abläufe nachvollziehen.

Bei dem vorliegenden Modell werden ausschließlich synchrone Methodenaufrufe verwendet. D.h. ein Methodenaufruf hat das Warten auf eine Antwortnachricht zur Folge. Bei der folgenden Diskussion einiger Realisierungsmöglichkeiten der browse-Methode in der Klasse TravelSchemeContainer wird in einer Version asynchroner Methodenaufruf beispielhaft modelliert. Die erste Variante in der Abbildung 153 ist unstrukturiert und schwer verständlich. Implementiert ist der Aufruf der show-Methode für jedes vom TravelSchemeContainer aggregierte TravelScheme-Objekt. Diese unstrukturierte Variante wird hier gezeigt, um zu verdeutlichen, daß auch bei der Verwendung von Petrinetzen zur Modellierung von Methoden die Gefahr besteht, unsauber zu programmieren. Zur Vermeidung unsauberer Modellierung von Methoden sollten wohldefinierte Kontrollkonstrukte eingesetzt werden. Einige sind im dritten Teil des Anhangs abgebildet.

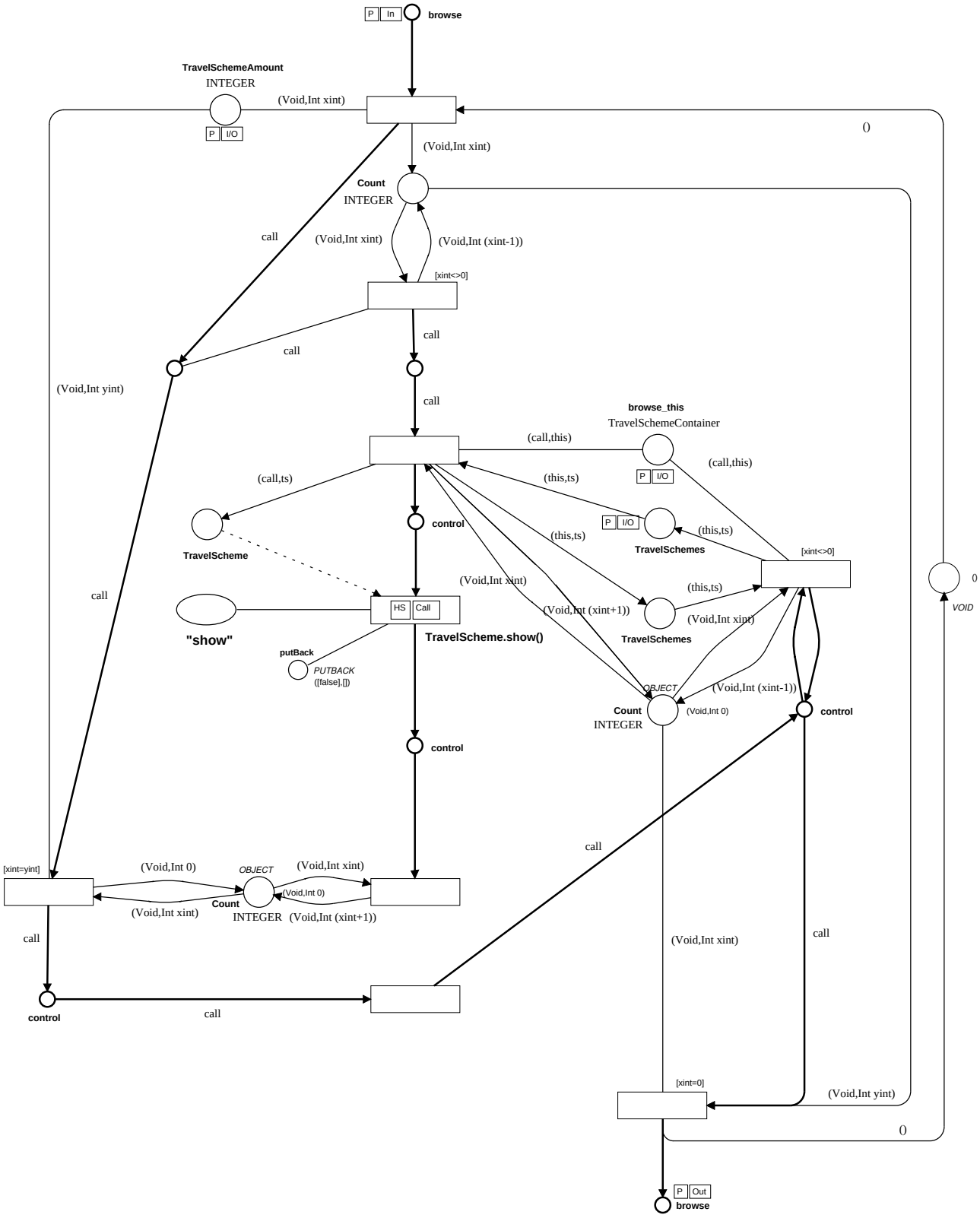


Abbildung 153: Unstrukturierte browse-Methode

In der Abbildung 154 ist die gleiche Methode übersichtlicher modelliert. Das Schalten der ersten Transition in dieser Methodenrealisierung bewirkt, daß die Marke mit der Anzahl der TravelSchemes von der Stelle ‘TravelSchemeAmount’ abgezogen wird. Dadurch können keine anderen Methoden zeitgleich ausgeführt werden, die die gleiche Marke, d.h. die Anzahl der TravelSchemes, benötigen.

Die Marke wird in der lokalen Stelle ‘DecrementCounter’ abgelegt und als Zähler verwendet. Dieser wird von der Transition heruntergezählt, die bei jedem Herausnehmen einer Objektreferenz aus der Stelle ‘TravelSchemes’ eine Nachricht mit dem Aufruf der show-Methode an das Objekt mit dieser Referenz sendet. Erst wenn alle Referenzen aus der Stelle herausgenommen wurden, wird die Zählermarke in die Stelle ‘IncrementCounter’ verschoben und nach jedem Aufruf einer show-Methode wird eine Objektreferenz zurück in die Stelle ‘TravelSchemes’ gelegt. Wenn für alle TravelSchemes die show-Methoden aufgerufen wurden ist der Wert der Zählermarke in der Stelle ‘IncrementCounter’ gleich der Originalanzahl in der lokalen Stelle ‘TravelSchemeAmount’ und die browse-Methode wird beendet. Dabei wird die Marke mit der Anzahl der TravelSchemes wieder auf die ursprüngliche Stelle zurückgelegt. Bei dieser Realisierung wird der Kontrollfluß nicht sauber verwendet, weil die Kontrollflußmarke zur Aktivierung der letzten Transition schon von Beginn an in deren Eingangsstelle liegt. Die Aktivierung der letzten Transition wird damit nur durch die Marke in der Stelle ‘IncrementCounter’ durchgeführt.

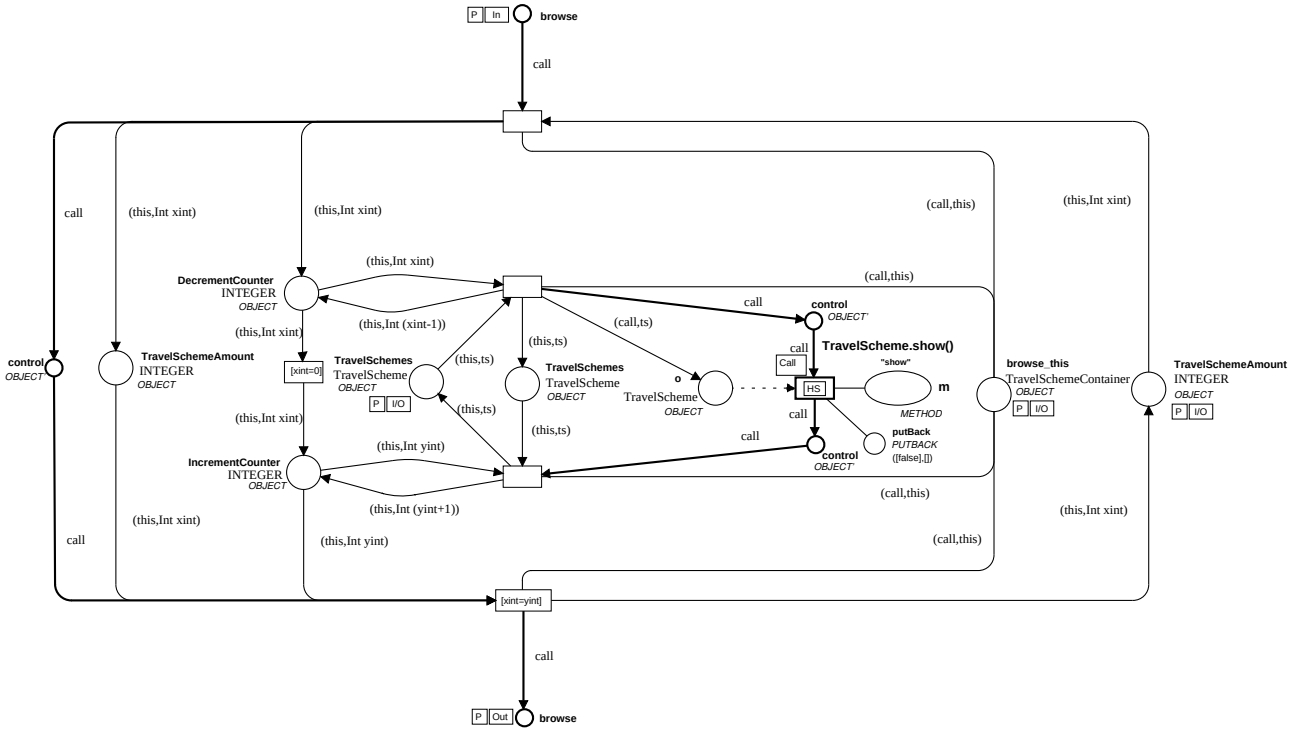


Abbildung 154: Strukturierte browse-Methode

In der Abbildung 155 wird der Kontrollfluß anders modelliert. Hier fällt er mit den für den Kontrollfluß entscheidenden Daten zusammen. Die letzte Transition in der browse-Methode wird nur durch diesen so modellierten Kontrollfluß aktiviert. Ein weiterer Unterschied zu der vorherigen Realisierung ist die Verwendung von ‘call’ statt ‘this’ in einigen Kantenbeschriftungen. Dadurch werden alle temporär erzeugten Marken für den individuellen Aufruf eingefärbt. Da diese Methode nicht nebenläufig zu sich selbst aufgerufen werden kann, ist dies hier nicht so entscheidend. Aber generell sollten Methoden so modelliert werden, um nicht die Nebenläufigkeit aus technischen Gründen auszuschließen.

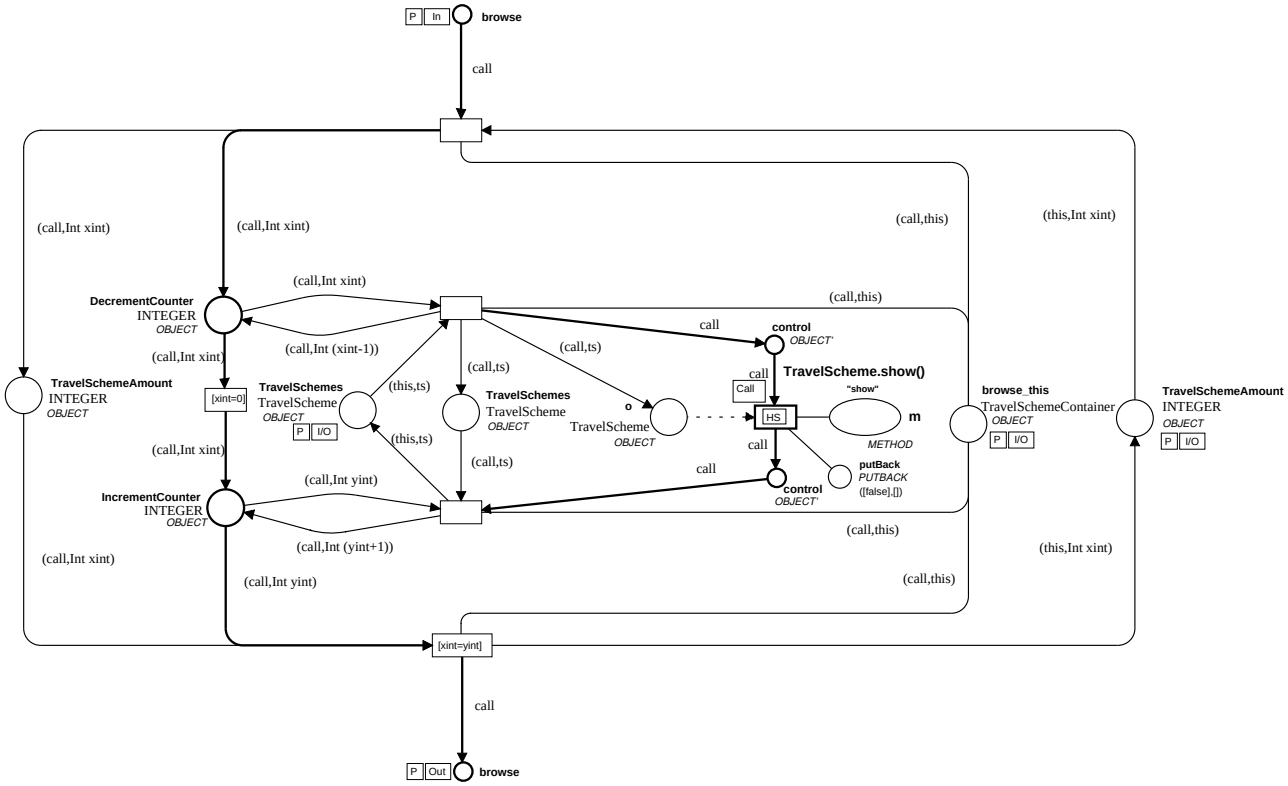


Abbildung 155: Korrigierter Kontrollfluß in der browse-Methode

Eine asynchrone Variante der browse-Methode wird in der Abbildung 156 dargestellt. Im Unterschied zu der synchronen Variante wird hier bei dem Zurücklegen der TravelSchemes aus der lokalen Stelle in die ursprügliche Stelle durch die inkrementierende Transition keine Kontrollmarke benötigt, die in der synchronen Version nach dem Aufruf der show-Methode für ein TravelScheme durch die inkrementierende Transition von der Ausgangsstelle der Substitutionstransition ‘Call’ abgezogen wurde. Das Beispiel zeigt wie einfach eine asynchrone Lösung implementiert werden kann.

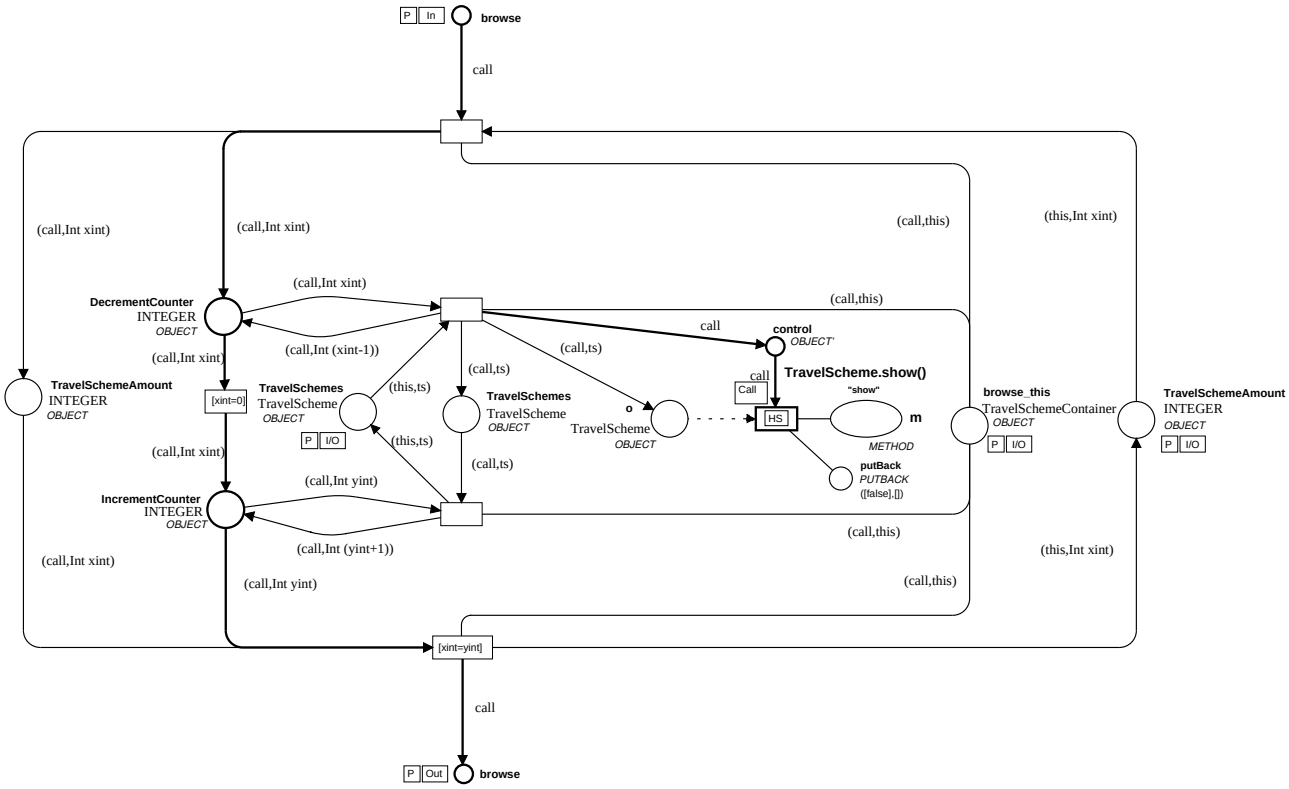


Abbildung 156: browse-Methode mit asynchronem Methodenaufruf

Als letzter Vorschlag zur Modellierung der browse-Methode wird die Variante mit nur einem Zähler aus der Abbildung 157 vorgestellt. In dieser synchronen Lösung wird der Kontrollfluß teilweise explizit Modelliert und teilweise mit dem Datenfluß zusammengelegt. Auch hier werden zunächst alle TravelSchemes aus der einen Stelle entnommen und dabei der Zähler heruntergezählt. Ist dieser auf ‘0’, so schaltet eine Transition die Kontrollflußmarke aus der oberen Stelle in die untere und steht der Transition zur Verfügung, die dafür sorgt, die TravelSchemes aus der lokalen Stelle in die Instanzvariablenstelle zurückzulegen. Erst wenn alle show-Aufrufe abgeschlossen wurden ist der Zähler auf den ursprünglichen Wert hochgezählt worden und die letzte Transition kann schalten und damit die browse-Methode sauber beenden.

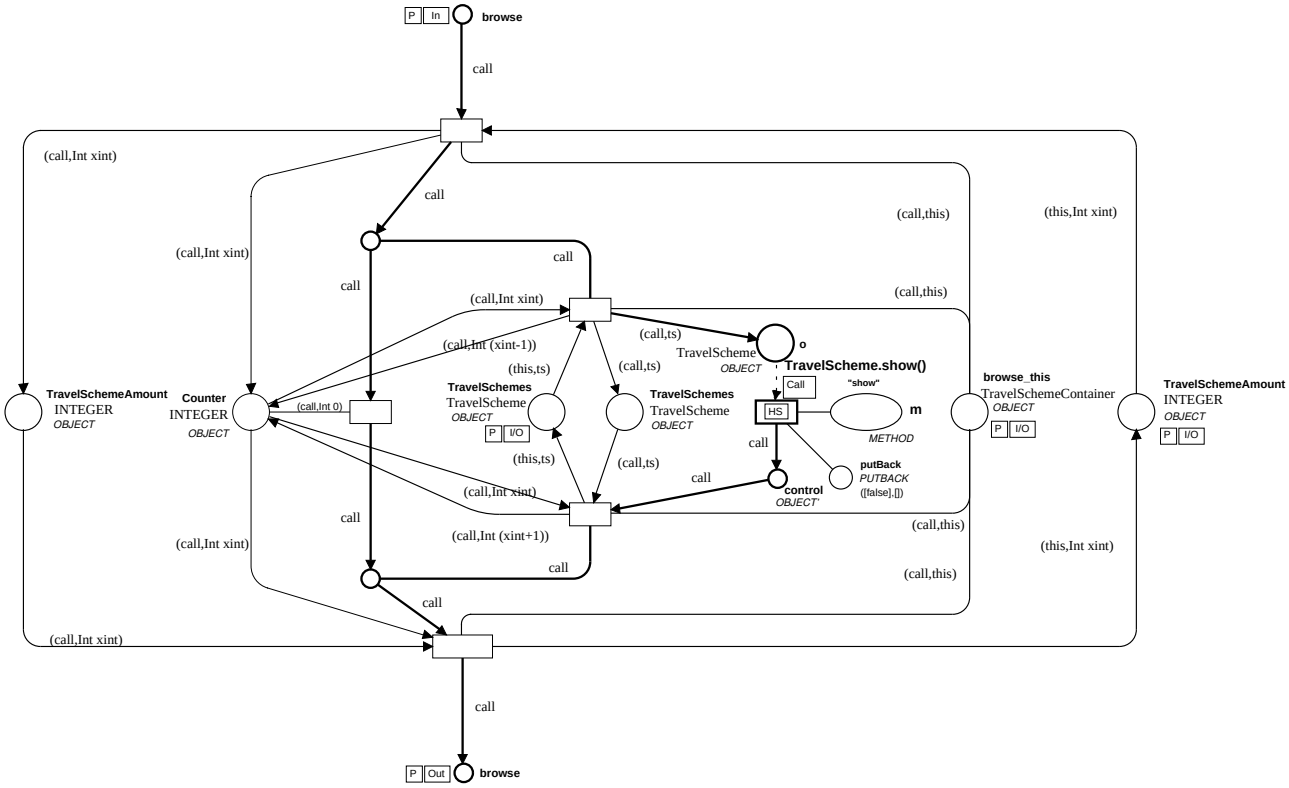


Abbildung 157: browse-Methode mit einem lokalen Zähler

5.4 Hierarchie des Petrinetzes

Die Netzarchitektur ist wesentlich durch die zur Modellierung der Petrinetze verwendete Software Design/CPN geprägt. Die Hierarchieseite des Minitours-Petrinetzes wird hier erläutert und weitere softwarespezifische Aspekte erklärt, sowie auf einige technische Simulationsaspekte eingegangen.

Die Hierarchieseite in der Abbildung 158 zeigt alle Seiten des Minitours-Petrinetzmodells. Dargestellt sind die hierarchischen Beziehungen zwischen den Seiten durch gerichtete Kanten. Eine Seite verwendet die Seite, zu der eine Kante zeigt, als Unterseite, d.h. eine Transition auf der Oberseite wird durch die Unterseite substituiert. Die Kante ist mit dem Namen der Substitutionstransition beschriftet, sofern dieser ein Name zugewiesen wurde. Bei den Interfaces, die als Unterseiten der Klassen verwendet werden, ist die Beschriftung einheitlich ‘implements’. Die Verwendung der Class-Seite als Unterseite wird nicht explizit durch eine Kantenbeschriftung dokumentiert. Die Transitionen, die durch die Call-Seite ersetzt werden, sind mit der aufgerufenen Methode beschriftet, d.h. somit auch die Kante auf der Hierarchieseite. Zum Beispiel wird auf Seite der Methode ‘defineTravelScheme()’ bei der Klasse ‘ManagerGUI’ eine Nachricht an die Klasse ‘TravelSchemeContainer’ mit dem Aufruf der Methode ‘createTravelScheme()’ mit Hilfe der Call-Seite erzeugt. Die Transition, die durch die Call-Seite verfeinert wird, hat den Namen ‘TSC.createTravelScheme()’. Dieses entspricht genau der Beschriftung der Kante von der Seite ‘ManagerGUI.defineTravelScheme()#27’ zur Call-Seite auf der Hierarchieseite. Mehrere Kantenbeschriftungen deuten auf entsprechend viele übereinanderliegende Kanten hin.

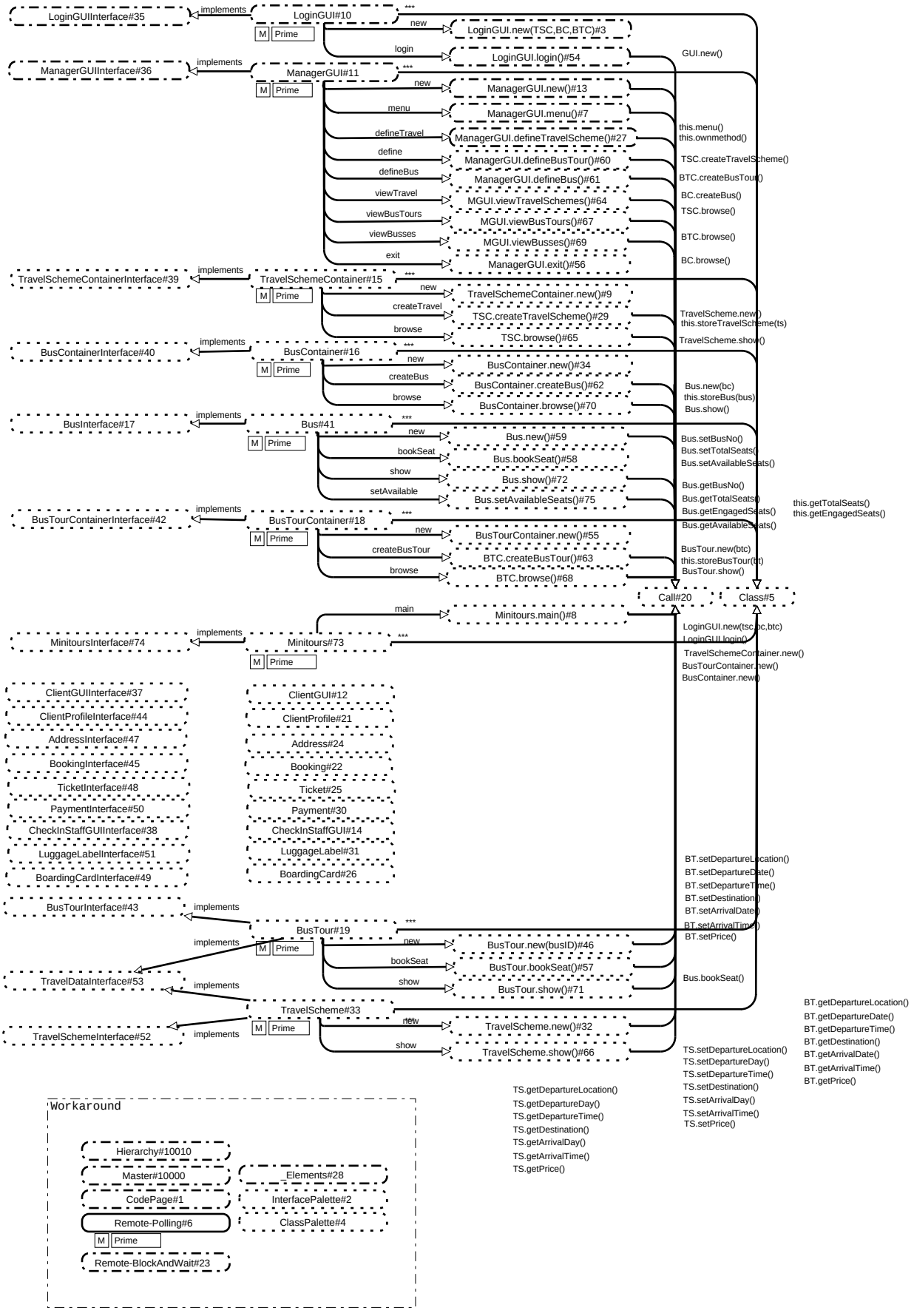


Abbildung 158: Hierarchieseite

Dadurch, daß nur Unterseiten mehrfach verwendet werden können, müssen Interfaces als Unterseiten modelliert werden. Bei der Betrachtung der Netze scheint es ungewöhnlich, daß die Interfaces Unterseiten der Klassen-Seiten sind. Intuitiv würde man erwarten, daß das Interface die oberste Seite ist und die Klassen-Seite als Unterseite hat, so wie die Klassen-Seiten die Methoden-Seiten als Unterseiten haben. Deshalb erscheint es etwas verwirrend, daß die Kontrollstellen auf der linken Seite der Interface-Seiten als Out-Stellen definiert sind und die Stellen auf der rechten Seite als In-Stellen.

Bei der Code-Page handelt es sich um eine technisch notwendige Seite. Auf dieser werden die Farben der Stellen definiert, im Netz verwendete Variablen deklariert und ML-Funktionen programmiert. Ein Beispiel für eine Farbdefinition ist ‘color METHOD = STRING’. Die als Typ ‘METHOD’ deklarierten Variablen können also Zeichenketten als Werte annehmen und Stellen können Marken dieses Typs enthalten. Die ML-Funktionen können im Netz als Kantenbeschriftungen oder in Code-Regionen verwendet werden. Der Aufruf der Funktion ‘GetString (cr_prompt:string, cr_def:string, canc.string)’ generiert ein Fenster mit einer Eingabeaufforderung.

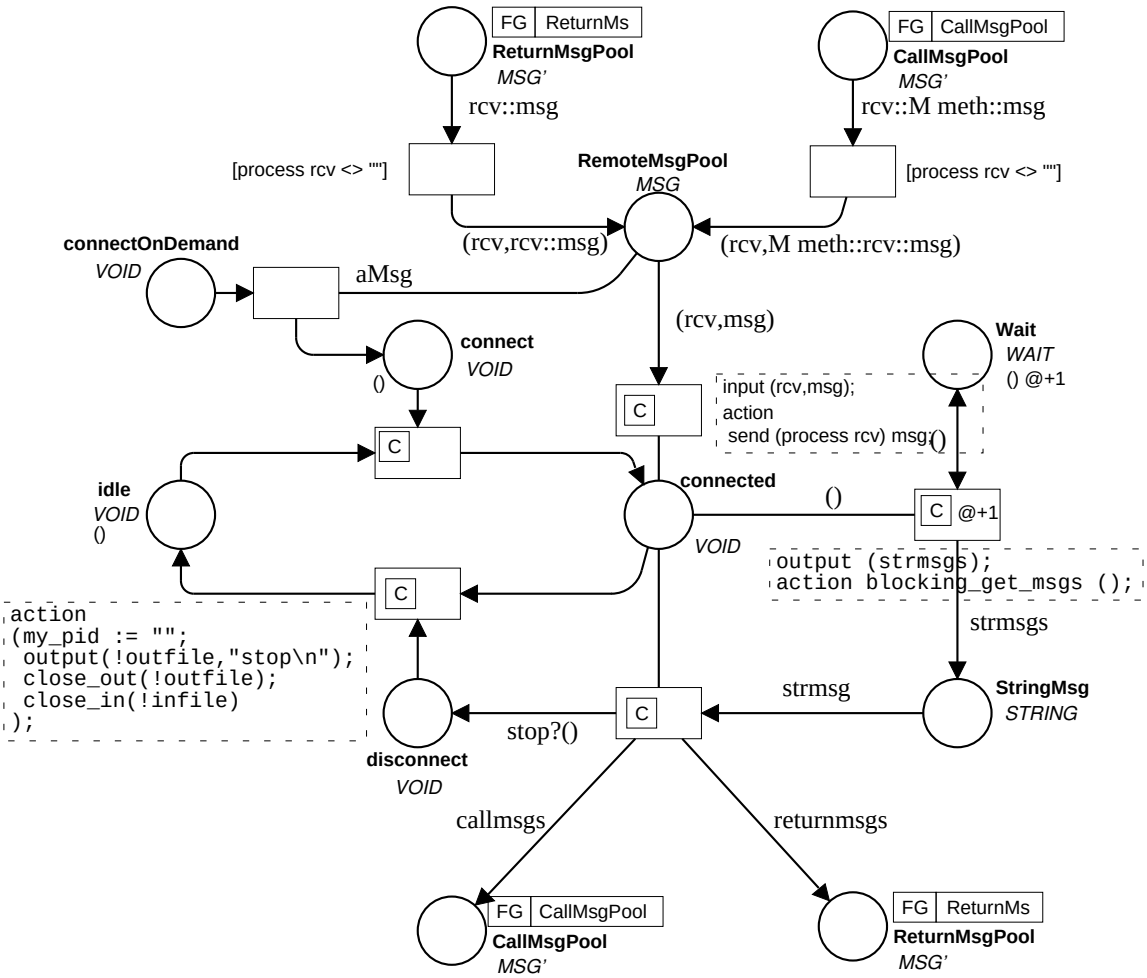
Aus Übersichtsgründen sind auf der Hierarchieseite nicht alle Interface- Klassen- und Methodenseiten abgebildet. Diese Seiten des Minitoursmodells können im zweiten Teil des Anhangs nachgeschlagen werden.

5.5 Ausführung der Spezifikation

Bevor die Spezifikation ausgeführt werden kann, wird die Syntax des Petrinetzes überprüft. Wenn alle Deklarationen auf der Code-Page korrekt sind und das Netz fehlerfrei ist, wird das Netz in eine simulationsfähige Datei übersetzt. Von den mit ‘M Prime’ gekennzeichneten Seiten werden Instanzen erzeugt und durch die Startnachricht in der Stelle ‘inMsg’ auf der Seite ‘Minitours#73’ ist die linke Transition auf der Seite MinitoursInterface#74’ aktiviert und schaltet, sodaß die Methode ‘main()’ der Klasse ‘Minitours’ aufgerufen wird.

Läuft die Simulation lokal in einem Prozeß, so findet die Kommunikation der Objekte über die Stellen ‘ReturnMsgPool’ und ‘CallMsgPool statt’. Bei der verteilten Simulation existiert für jeden Design/CPN-Programmaufruf ein eigener Prozeß (Messenger). Die Objekte können sich auf verschiedenen Rechnern befinden. Jedes in einem Prozeß laufende Petrinetz benötigt die Remote-Seite aus der Abbildung 159, um die Nachrichten an die richtigen Objekte senden zu können. Die Kommunikation mit der auf einem der Remote-Seite bekannten Rechner aktiven Mailbox übernimmt der Messenger. In [KMW 98] wird diese Kommunikation ausführlich vorgestellt. Es werden dort verschiedene Realisierungen diskutiert, der Aufruf von Java-Objekten erläutert und die Vorteile des vorgestellten Ansatzes in Beispielanwendungen aufgezeigt. Die in [KMW 98] vorgestellte Remote-Seite verwendet einen Polling-Mechanismus, um zu überprüfen, ob neue Nachrichten vorliegen. In der Abbildung 159 ist ein Block-and-Wait-Mechanismus implementiert, der nicht so rechenintensiv ist wie die Polling-Lösung.

```
action
 let val name = "GUIProcess";
  val (inf,outf) = my_execute("/home/tgi_3/tgi3/bin/messenger",[name]);
 in (my_pid := name; infile := inf; outfile := outf) end;
```

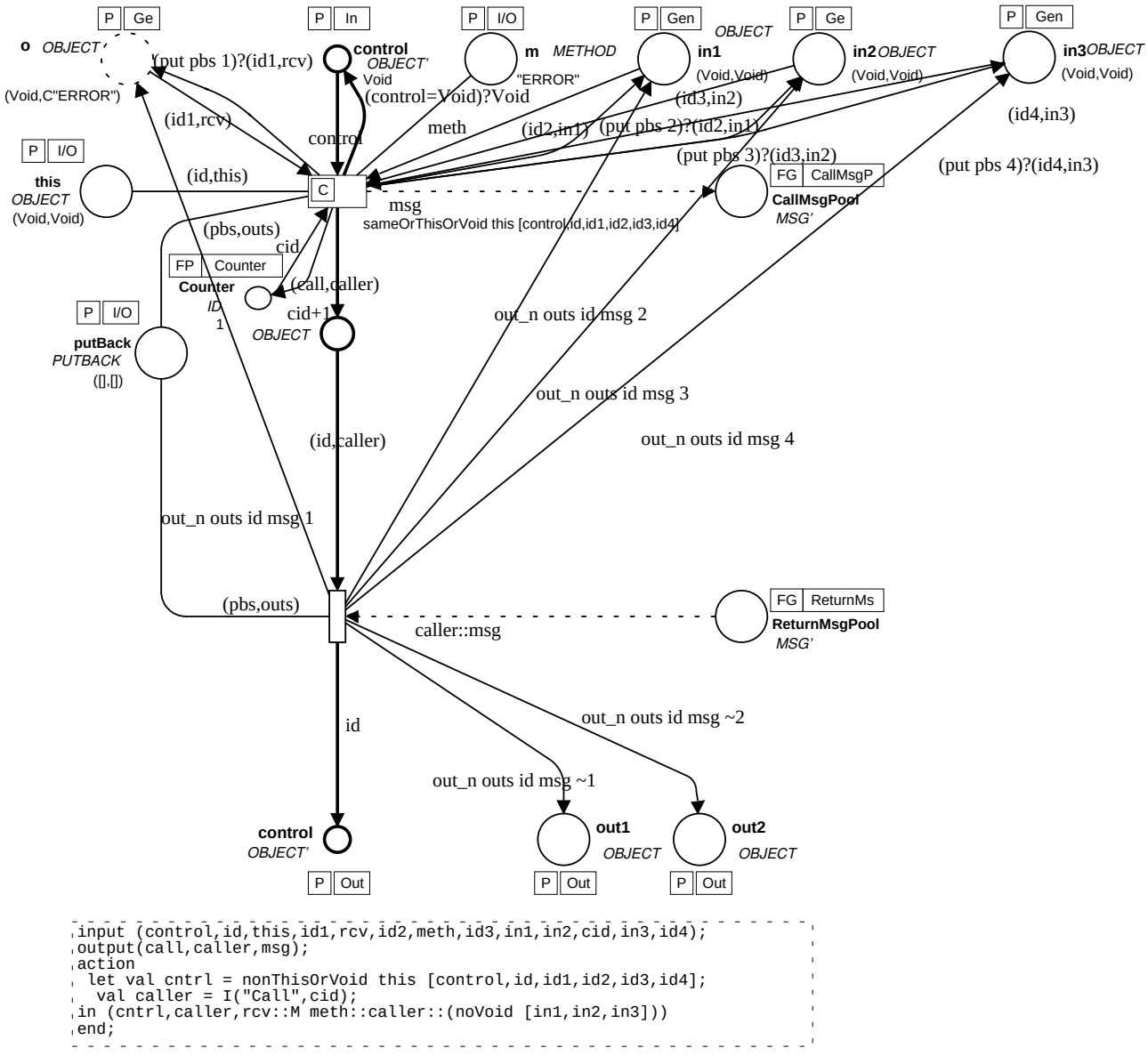



```
input (strmsg);
output (callmsgs,returnmsgs,stop);
action
  case toLocal (!my_pid) (strToMsg (strmsg)) of
     M(m)::rcv::param => (1‘(rcv::M(m)::param), empty, false)
   | I(i)::msg        => (empty,        1‘(I(i)::msg), false)
   | msg              => (empty,        1‘msg,         true);
```

Abbildung 159: Remote-Seite

Methodenaufrufe werden als Nachrichten auf der Call-Seite (Abbildung 160) in die Global-Fusion-Stelle ‘CallMsgPool’ gelegt. Handelt es sich um den Aufruf einer Methode in dem lokalen Design/CPN-Prozeß, so wird auf der Remote-Seite festgestellt, daß kein Prozeßname in der Nachricht enthalten ist. Die Kommunikation verläuft dann, wie oben erwähnt, nur über die globalen Fusions-Stellen. Ist der Prozeßname ungleich der leeren Zeichenkette, so handelt es sich um einen Methodenaufruf bei einem Objekt, das in einem anderen Prozeß läuft. Die Nachrichtenmarke wird aus der Stelle ‘CallMsgPool’ in die Stelle ‘RemoteMsgPool’ gelegt. Sofern der vorliegende Design/CPN-Prozeß mit dem Messenger verbunden ist, d.h. in der Stelle ‘connected’ eine Marke liegt, wird die send-Funktion in der Code-Region aufgerufen. Die mit ‘@+1’ beschriftete Transition schaltet, nachdem alle anderen, aktivierten Transitionen auf der Remote-Seite geschaltet haben. Bei diesem Schalten wird die Funktion ‘blocking_get_msgs()’ in der Code-Region ausgeführt. Hier findet die Kom-

munikation mit der Mailbox statt. Nachrichten an Objekte, die in diesem Prozeß aktiv sind, werden aus der Mailbox entnommen und Nachrichten an nichtlokale Objekte hineingelegt.




```
input (control,id,this,id1,rcv,id2,meth,id3,in1,in2,cid,in3,id4);
output(call,caller,msg);
action
  let val cntrl = nonThisOrVoid this [control,id,id1,id2,id3,id4];
    val caller = I("Call",cid);
in (cntrl,caller,rcv::M meth::caller::(noVoid [in1,in2,in3]))
end;
```

Abbildung 160: Call-Seite

5.6 Ergebnisse und Bewertung

Die Ergebnisse der Modellierung werden in diesem Abschnitt abschließend betrachtet. Eingegangen wird hier auf die Petrinetze als eingesetzte Modellierungstechnik, die gewählte Vorgehensweise und die zur Modellierung verwendeten Werkzeuge.

Die Untersuchung der objektorientierten Petrinetze zur Entwicklung einer ausführbaren Systemspezifikation hat zeigt, daß sich Petrinetze sehr gut zur Beschreibung der dynamischen Aspekte eines Systems eignen. Durch die Verwendung der objektorientierten Konzepte für die Petrinetze werden die Daten und der Zugriff auf diese Daten sauber modelliert. Die resultierenden Netze können leicht mit anderen objektorientierten Techniken kombiniert werden und stellen somit eine gute Ergänzung

zu den Techniken der Unified Modeling Language dar. In den folgenden Absätzen wird auf dieses Zusammenwirken einiger Techniken der objektorientierten Analyse und der objektorientierten Petrinetze eingegangen.

Die Klassennetze lassen sich direkt aus den Klassendiagrammen ableiten. Dadurch sind die Methodensignaturen eindeutig definiert. Bei der zur Erstellung einer Systemspezifikation des Reiseunternehmens gewählten Petrinetz-Realisierung wird eine Klasse durch mindestens eine Interface-Seite, eine Klassen-Seite und i.a. mehrere Methoden-Seiten modelliert. Klassen dienen als Erzeugungsmuster für Objekte. Die Attribute (Instanzvariablen) werden als Stellen auf der Klassen-Seite modelliert und die Methoden als Transitionen. Dies erlaubt eine übersichtliche Visualisierung der Klassen bzw. Objekte. Objekte eines Typs werden alle durch ein Petrinetz dargestellt. Die Attributwerte der unterschiedlichen Objekte werden durch die Kombination von Wert und zugehörender Objektidentifikation den Objekten eindeutig zugeordnet. Durch diese Faltung vieler Objekte in ein Petrinetz wird die grafische Repräsentation optimiert. Möglich ist dies, weil die Objekte einer Klasse alle die gleichen Attribute und Methoden besitzen. Der Aufruf einer Methode eines Objekts wird durch eine Aufrufidentifikation eindeutig gekennzeichnet. Somit ist gewährleistet, daß bei der Faltung der Objekte für jedes Objekt die Methoden individuell aufrufbar sind.

Die Methoden-Transitionen werden auf Unterseiten verfeinert. Speziell bei der Implementation der Methoden kommen die Möglichkeiten zur Modellierung von Nebenläufigkeit zum Einsatz. Für die Modellierung werden in dieser Arbeit einige Kontrollkonstrukte vorgeschlagen. Mit diesen lassen sich Sequentialität, Nebenläufigkeit, Entscheidungen, Schleifen, synchroner und asynchroner Methodenaufruf und wechselseitiger Ausschluß übersichtlich darstellen. Durch diese Visualisierung der Methoden kann die modellierte Funktionalität sehr schnell erfaßt werden. Die Methoden-Netze bieten damit eine gute Möglichkeit zur Diskussion des modellierten Verhaltens.

Der Methodenaufruf erfolgt durch Nachrichtenaustausch zwischen den Objekten. Eine Nachricht wird auf der Interface-Seite so in ihre Bestandteile zerlegt, daß der aufgerufenen Methode auf der Klassen-Seite die benötigten Parameter zur Verfügung gestellt werden. Die Verbindung der Interface-Seite und der Klassen-Seite erfolgt über gemeinsame Stellen. Dieses Konzept ermöglicht die mehrfache Verwendung einer Interface-Seite für unterschiedliche Klassen. Die Methoden-Transitionen auf der Klassen-Seite sind nach dem gleichen Prinzip mit den Methoden-Seiten verbunden. Dies ermöglicht die mehrfache Verwendung der Methoden-Seiten durch unterschiedliche Klassen. Durch diese Verwendung des Hierarchiekonzeptes werden zwei Arten der Vererbung realisiert. Auf der einen Seite ist dies die Implementation mehrerer Interfaces und auf der anderen Seite die Code-Wiederverwendung. Mehrfachvererbung ist, analog zu dem Vererbungskonzept in der Programmiersprache Java, nicht vorgesehen. Nur Quasi-Mehrfachvererbung kann über die erstgenannte Implementation mehrerer Interfaces modelliert werden. Die Vererbung von Assoziationen ist im Rahmen dieser Arbeit nicht untersucht worden.

Bei den in dieser Arbeit zur Modellierung eingesetzten Petrinetzen erfolgt der Methodenaufruf synchron. Das bedeutet, daß zu einer Nachricht mit einem Methodenaufruf auf eine Antwortnachricht gewartet werden muß, bevor die Abarbeitung der aufrufenden Methode fortgesetzt werden kann. Der Vorteil liegt in dem einfachen Format der Nachrichten und der einfachen Kommunikation. Eine Nachricht zum Aufruf einer Methode besteht aus Sender, Empfänger, Methodenname und Parameterliste. Eine Antwortnachricht besteht nur aus Empfänger und Parameterliste. Asynchrone Methodenaufrufe können bei der Verwendung dieses Nachrichtenformats realisiert werden, solange die aufgerufene Methode keine Parameter zurückliefert. Das Warten auf die Antwortnachricht wird in diesem Fall einfach unterbunden. Liefert beim asynchronen Methodenaufruf die aufgerufene Methode einen Wert zurück, so muß das Nachrichtenformat der Antwortnachricht um den Sender ergänzt werden, damit diese Nachricht richtig zugeordnet werden kann. Im Fallbeispiel wird diese Art der

asynchronen Kommunikation nicht realisiert.

Während sich die Attribute und Methoden der Klassen gut auf die Elemente der Petrinetze übertragen lassen, erfordert die Umsetzung der Assoziationen die Modellierung komplexer Protokolle. In dieser Arbeit werden diese nicht alle explizit realisiert, sondern teilweise in die Klassen integriert. Bei einer Muß-Beziehung wird bei dem Aufruf der Konstruktor-Methode eines Objekts die Referenz auf die Klasse, zu der die Muß-Beziehung besteht, als Parameter übergeben. Die Realisierung der Muß-Beziehungen wird in dieser Arbeit durch die Verwendung eines Objektbezeichners simuliert. Bei der Erzeugung eines Objekts, das in einer Muß-Beziehung steht, wird der Objektbezeichner des anderen benötigten Objekts, nicht dessen Referenz, als Parameter angegeben. Das impliziert, daß das Objekt, dessen Objektbezeichner übergeben wird, existieren muß. Die Überprüfung, ob ein Objekt mit dem angegebenen Objektbezeichner existiert, ist in diesem Fallbeispiel nicht implementiert, weil die Realisierung der Protokolle für Assoziationen den Rahmen dieser Arbeit sprengen würde. Wenn im Fallbeispiel bei der Erzeugung einer Busreise ein Bus angegeben werden muß, ist der Manager dafür verantwortlich, eine Busnummer eines existierenden Busses einzugeben. Da das Löschen eines Busses im System nicht vorgesehen ist, wird der Aspekt der Assoziation, daß ein Bus, der für eine Busreise eingesetzt wird, nicht gelöscht werden darf, nicht berücksichtigt. Untersucht wird die Realisierung von Assoziationen als Petrinetze in [Rölke 99], deren Ergebnisse leider nicht rechtzeitig für diese Arbeit zur Verfügung standen.

Die Aggregationsbeziehung wird durch die explizite Modellierung des benötigten Protokolls realisiert. Die aggregierenden Objekte rufen den Konstruktor der aggregierten Objekte auf und erhalten deren Referenz als Rückgabewert. Diese Referenz speichern sie in einer speziellen Stelle. Das Löschen eines aggregierten Objekts wird nicht modelliert, könnte aber durch eine weitere Methode beim aggregierenden Objekt einfach hinzugefügt werden. Da im Klassendiagramm spezielle Container-Klassen verwendet werden, ist die explizite Modellierung der Aggregationsbeziehung eine gute Lösung. Die Verlagerung der anderen Assoziationen in die Klassen schränkt die Wiederverwendbarkeit der Klassen ein und das System wird unflexibler gegenüber Änderungen. Bei der Modellierung der Assoziationen als eigene Klassen wird das Petrinetz komplexer als bei der Integration der Assoziationen in die Klassen.

Das angewendete Vorgehen zur Entwicklung der ausführbaren Spezifikation hat sich im Rahmen der technischen Voraussetzungen bewährt. Mit den Techniken der Unified Modeling Language und dem Werkzeug ‘Rational Rose’ läßt sich sehr gut ein erstes Analysemodell erstellen. Mit den Use-Cases werden die Leistungen des Systems beschrieben. Das Klassendiagramm und die Interaktionsdiagramme werden als Grundlage für die Entwicklung der Petrinetze verwendet. Die Zustandsdiagramme können als Hilfe bei der Modellierung der Methoden verwendet werden. Explizit vorgesehen ist letzteres bei dem vorgeschlagenen Vorgehen nicht, weil sich die Petrinetze sehr gut zur Modellierung von Methoden eignen. Prototyping bei dem Entwurf der Netze erhöht die Kommunikationsmöglichkeiten mit dem Anwender und somit dessen bessere Integration in den Entwicklungsprozeß. Durch die Simulation, d.h. Ausführbarkeit der Netze, können Szenarien überprüft und damit das System getestet werden. Die Entwicklung von evolutionären Prototypen bietet den Vorteil der Weiterverwendung von korrekt modellierten Systemteilen und die Ersetzung bzw. Erweiterung von falsch bzw. noch nicht modellierten Teilen.

Zur Modellierung der Petrinetze wird als Werkzeug ‘Design/CPN – Version 3.02’ eingesetzt. Das Programm eignet sich sehr gut zum Erstellen und Simulieren von Petrinetzen und hat bei einem Vergleich mehrerer Petrinetzwerkzeuge in [Störrle 98] die beste Beurteilung bekommen. Dennoch ist die Erstellung von objektorientierten Petrinetzen sehr aufwendig, weil der Entwurf dieser Art Petrinetze von Design/CPN nicht unterstützt wird. Aus diesem Grund ist die modellierte Systemfunktionalität auf die wesentlichen Aspekte des Fallbeispiels reduziert.

Das Überprüfen der Syntax und das Übersetzen des Petrinetzes in die ausführbare Version dauert mit zunehmender Größe des Minitours-Prototypen sehr lange. Beschleunigt werden diese Vorgänge durch den Einsatz der Möglichkeiten aus [KMW 98]. Die Klassen werden separat in jeweils überschaubaren Teilmodellen als Petrinetze entwickelt. Durch die Möglichkeit der verteilten Simulation bei der Ausführung des Prototypen, können diese Modellteile relativ schnell überprüft und übersetzt werden. In dieser Arbeit wurden die Möglichkeiten zur verteilten Simulation erstmals in größerem Umfang, zeitgleich zu einer stattfindenden Lehrveranstaltung, eingesetzt. Bei der bisherigen technischen Realisierung wird allerdings keine gemeinsame Hierarchieseite erstellt. Möglichkeiten hierzu eröffnen die Programme zur Speicherung der Petrinetze im Textformat aus [Maier 98] und [Lyngsø 98].

Durch die in [KMW 98] vorgestellte Mailbox-Kommunikation können die objektorientierten Petrinetze auch mit Objekten interagieren, die in anderen Programmiersprachen implementiert sind. Bestehende Klassenbibliotheken, z.B. für Benutzeroberflächen oder Datenbankanbindungen, brauchen nicht neu entwickelt zu werden. Die Entwickler können sich auf die Modellierung der Methoden konzentrieren, mit denen die Abläufe im Anwendungssystem realisiert werden. In dieser Arbeit wurde diese Möglichkeit nicht verwendet, da die Modellierung der Klassen als objektorientierte Petrinetze im Fokus der Untersuchung stand und nicht die Gestaltung einer Benutzerschnittstelle.

Die folgenden Probleme bei der Entwicklung einer ausführbaren Systemspezifikation mit gefärbten Petrinetzen und der Verwendung eines prozeßorientierten, funktionalen Vorgehens aus [Netzebandt 97] wurden gelöst: Durch den Einsatz der objektorientierten Petrinetze wurden die in [Netzebandt 97] stark verzahnten Systemfunktionen entkoppelt. Damit wird eine hohe Flexibilität bei der Änderung des Systemverhaltens während des Prototypings erreicht und die Wiederverwendbarkeit erhöht.

6 ZUSAMMENFASSUNG UND AUSBLICK

Die Zusammenfassung bietet einen Überblick über die vorliegende Arbeit und der Ausblick zeigt mögliche neue Projekte auf.

6.1 Zusammenfassung

In dieser Arbeit werden Ansätze zur Entwicklung einer ausführbaren Systemspezifikation mit objektorientierten Petrinetzen vorgestellt und einer dieser Ansätze an einem Fallbeispiel erprobt. Motiviert wurde diese Arbeit u.a. durch die bei der Spezifikation eines Reiseunternehmens in [Netzebandt 97] aufgetretenen Probleme. Dort wurde das System durch funktionale Dekomposition in überschaubare Funktionen aufgeteilt und diese unter Betrachtung von Szenarien durch Petrinetze implementiert und getestet. Diese sogenannten Szenariennetze wurden schrittweise so erstellt, daß jedes Szenariennetz ein ausführbarer Prototyp ist. Die Probleme bei der Modellierung des Systems sind in der unflexiblen Struktur der Petrinetz-Prototypen begründet. Die Reihenfolge der spezifizierten Abläufe läßt sich nicht einfach ändern, das Petrinetz ist extrem systemspezifisch und die Wiederverwendbarkeit von Komponenten ist sehr stark eingeschränkt.

Diese Arbeit liefert Lösungsansätze für diese Probleme durch die Kombination der Vorteile der Objektorientierung und der Petrinetze. Die Objektorientierung erhöht den Grad der Wiederverwendbarkeit von Software und verbessert die Flexibilität bezüglich der Anpassung von Anforderungsänderungen in der bereits erstellten Spezifikation. Petrinetze hingegen eignen sich zur Modellierung der dynamischen Systemaspekte und besonders zur adäquaten Darstellung von Nebenläufigkeit. Sie ermöglichen die Modellierung von ausführbaren Spezifikationen und können potentiell zur Verifikation von Systemen verwendet werden.

Bei der Entwicklung des Netzes in dieser Arbeit wurde ausgehend von dem Analysemodell des Reiseunternehmens in UML-Notation jede Klasse und deren Interfaces und Methoden als Petrinetz modelliert. Auch bei dieser Vorgehensweise wurden die Methoden der Klassen schrittweise so implementiert, daß die Petrinetz-Prototypen stets ausführbar sind. Als komplex hat sich die Modellierung der Muß-Beziehungen zwischen Klassen erwiesen. Die Implementation der hierzu benötigten Protokolle lenkten von der eigentlichen Spezifikation des Systems ab. Die Assoziationen wurden aus Gründen der zunehmenden Größe des Netzes nicht als eigene Klassen modelliert, wie dies in einem der vorgestellten Ansätze vorgeschlagen wird, sondern sind ansatzweise in die Klassen integriert. Dies hat zur Folge, daß zwar das Netz etwas kleiner geblieben ist als bei der expliziten Modellierung der Assoziationen, aber auch unflexibler bezüglich der Änderung von Beziehungen zwischen Klassen. Die Anpassung der Funktionalität an geänderte Anforderungen hingegen wird durch die objektorientierte Modellierung unterstützt und die Simulation des Petrinetzes ermöglicht die sofortige Überprüfung der Änderungen.

Da diese Modellierung z.Zt. durch kein Werkzeug unterstützt wird, ist die Erstellung der Petrinetz-Prototypen noch sehr aufwendig. Das vorgestellte Petrinetz ist mit über 1500 Stellen und über 300 Transitionen eines der größten bisher am Arbeitsbereich erstellten Petrinetze. Dieses ist für das gewählte Beispiel relativ groß und zeigt, daß der Einsatz eines Werkzeugs zur direkten Unterstützung der in dieser Arbeit vorgestellten Konzepte zwingend erforderlich ist. Ein solches Werkzeug zur Generierung von objektorientierten Petrinetzen aus Klassendiagrammen wird am Arbeitsbereich TGI derzeit entwickelt und wird in Zukunft die Assoziationen und Methodenschnittstellen implementieren. Der Entwickler kann sich dann auf die eigentliche Systemspezifikation konzentrieren und die vorgeschlagenen Kurzschreibweisen bei der Modellierung verwenden.

6.2 Ausblick

Die Lösungen der Probleme, die diese Arbeit motiviert haben, haben neue Fragestellungen aufgeworfen. Diese werden in diesem Ausblick hinsichtlich möglicher weiterführender Projekte aufgezeigt.

Die komplexen Protokolle bei der Modellierung der Assoziationen zwischen Klassen sollten automatisch implementiert und bei der Simulation ausgeführt werden. Der Entwickler sollte nicht mit der Modellierung der Assoziationen belastet werden. Die Realisierung der Assoziationen als eigene Klassen bietet die Möglichkeit, die auszuführenden Aktionen bei der Erzeugung und dem Löschen von Objekten, die in dieser Assoziation zueinander stehen, entkoppelt von den Klassen der Anwendung zu implementieren. Behandelt wird dieses Thema bereits in [Rölke 99].

Ein weiterer Ansatz zur Entlastung des Entwicklers ist die automatische Modellierung von wechselseitigem Ausschluß, Synchronisation und Asynchronität und ähnlichen Protokollen als Beziehungen zwischen Methoden. Dafür müßte eine Möglichkeit zur Beschreibung dieser Beziehungen entwickelt werden und ein Generator zur Implementation der Protokolle. Diese könnten Teil der Klassen-Seite sein, denn dort befinden sich die Methoden auf der obersten Hierarchieebene.

Mechanismen zur Ausnahmebehandlung sind bisher nicht implementiert und könnten in einer Weiterentwicklung dieses Beispiels Fokus der Modellierung sein.

Ein weiterer Schwerpunkt könnte auf der Entwicklung einer grafischen Benutzeroberfläche liegen, die in der Programmiersprache Java implementiert wird und mit den Objektnetzen kommuniziert.

Zur Realisierung der Kommunikation zwischen verschiedenen Arten von Objektnetzen könnte ein Nachrichtentransformator implementiert werden.

Ein Werkzeug zur Generierung von effizient ausführbarem Code könnte die Spezifikation direkt in eine verwendbare Anwendungssoftware umsetzen. Ansätze zur Code-Generierung findet man unter der Internet-Adresse ‘www.daimi.au.dk/designCPN’.

Die Arbeit des Entwicklers sollte sich nur auf die Definition von Workflows und die Implementation der Methoden als Petrinetze beschränken. Hierzu müßte eine geeignete Oberfläche entwickelt werden. Teile der objektorientierten Petrinetze könnten, wie oben beschrieben, automatisch generiert werden. Außerdem wird ein Werkzeug benötigt, daß es erlaubt, die abkürzenden Schreibweisen bei der Modellierung zu verwenden, welche dann automatisch übersetzt werden.

Die Möglichkeiten der verteilten Simulation könnten zu einer Architektur zur verteilten Entwicklung von ausführbaren Systemspezifikationen mit Petrinetzen ausgebaut werden.

Für Petrinetze gibt es noch viele Anwendungsfelder, in denen dynamische Abläufe modelliert werden. Dazu gehören auch Steuerungs- und Prozeßleitaufgaben. Der Einsatz von objektorientierten Petrinetzen in diesem Bereich ist sicherlich eine spannende Herausforderung.

Ein z.Zt. aktuelles Feld ist das der agentenorientierten Programmierung. Die in dieser Arbeit vorgestellten Konzepte aus dem Bereich der objektorientierten Systementwicklung könnten zur Modellierung von Agenten weiterentwickelt werden. Ansätze zur agentenorientierten Analyse mit Petrinetzen findet man unter der Internetadresse ‘www.aoa.de’.

7 LITERATURANGABEN

[Balzert 96] Balzert, H.: Methoden der objektorientierten Systemanalyse, Spektrum Verlag, 1996

[Becker und Moldt 93a] Becker, U.; Moldt, D.: Objektorientierte Konzepte für gefärbte Petrinetze, In [Scheschonk und Reisig 93], S. 140-151

[Becker und Moldt 93b] Becker, U.; Moldt, D.: Object-Oriented Concepts for Coloured Petri Nets, In IEEE 'Conference Proceedings, IEEE International Conference on Systems, Man and Cybernetics', Band 3, S. 279-286, Frankreich, 1993

[Booch 91] Booch, G.: Object-Oriented Design with Applications, The Benjamin/Cummings Publishing Company, Redwood City, 1991

[Booch 94] Booch, G.: Object-Oriented Analysis and Design with Applications, The Benjamin/Cummings Publishing Company, Redwood City, 1994

[Bruno 94] Bruno, G.: Model-based Software Engineering, Chapman & Hall, London, 1994

[Buchs und Guelfi 91] Buchs, D.; Guelfi, N.: CO-OPN: A Concurrent Object Oriented Petri Net Approach, In 'Application and Theory of Petri Nets, 12th International Conference, Gjern, Denmark', University of Aarhus, IBM Deutschland, 1991

[Burkhardt 97] Burkhardt, R.: Objektorientierte Modellierung für die Praxis, Addison-Wesley, 1997

[Christensen und Hansen 93] Christensen, S.; Hansen, N. D.: Coloured Petri Nets Extended with Place Capacities, Test Arcs and Inhibitor Arcs, In [Marsan 93]

[Coad/Yourdon 90] Coad, P.; Yourdon, E.: Object-Oriented Analysis, 2.Auflage, Yourdon Press, Prentice Hall, Englewood Cliffs, 1991

[Coleman 94] Coleman, D.: Object-Oriented Development: The Fusion Method, Prentice Hall Englewood Cliffs, NJ, 1994

[Cornell 97] Cornell, G.; Horstmann, C. S.: Core Java 2nd edition; Prentice Hall, 1997

[DeMarco 79] DeMarco, T.: Structured Analysis and System Specification, Yourdon Press, Prentice Hall, Englewood Cliffs, 1979

[Design/CPN - Tut 93] Meta Software Corporation, Cambridge, MA, USA: Design/CPN Tutorial, Version 2.0, 1993

[Design/CPN - Man 93] Meta Software Corporation, Cambridge, MA, USA: Design/CPN Reference Manual, Version 2.0, 1993

[Dijkstra 72] Dijkstra, E. W. ; Dahl, O. J.; Hoare, C. A.: Structured programming, Acad. Pr., London, 1972

[Fowler 97] Fowler, M.; Scott, K.: UML Distilled, Addison-Wesley Longman, 1997

[Hatley 87] Hatley, D.; Pirbhai, I.: Strategies for Real-Time Specification, Dorset House Publishing, 1987

[Jacobson 92] Jacobson, I.: Object-oriented software engineering - a use case driven approach, Addison-Wesley [u.a.], Wokingham, 1992

[Jensen 92] Jensen, K.: Coloured Petri Nets: Basic Concepts, Analysis Methods and Practical Use, Volume 1, Springer Verlag Berlin Heidelberg, 1992

[Jensen 94] Jensen, K.: Coloured Petri Nets: Volume 2, Analysis Methods, EATCS Monographs on Theoretical Computer Science, Springer Verlag, Berlin Heidelberg New York, 1994

[Jensen 97] Jensen, K.: Coloured Petri Nets – Basic Concepts, Analysis Methods and Practical Use, Volume 3, Springer Verlag, 1997

[Jensen 98] Jensen, K.: Workshop on Practical Use of Coloured Petri Nets and Design/CPN, Aarhus, Denmark, 10-12 June 1998

[Jessen und Valk 87] Jessen, E.; Valk, R.: Rechensysteme: Grundlagen der Modellbildung, Springer Verlag Berlin Heidelberg, 1987

[Kahlbrandt 98] Kahlbrandt, B.: Software-Engineering – objektorientierte Software-Entwicklung mit der Unified Modeling Language, Springer Verlag, 1998

[KMW 98] Kummer, O.; Moldt, D.; Wienberg, F.: A Framework for Interacting Design/CPN- and Java-Processes, Universität Hamburg, Fachbereich Informatik, 1998

[Krauß 96] Krauß, T.-O.: Objekt Orientierte Petri Netze (OOPN) – Ein Vergleich aktueller Modelle, Studienarbeit, Universität Hamburg, Fachbereich Informatik, 1996

[Lakos 94] Lakos, C.A.: Object Petri Nets - Definition and Relationship to Coloured Petri Nets, Bericht R-94-3, Department of Computer Science, University of Tasmania, GPO Box 252C Hobart Tasmania 7001, April 1994

[Lakos 95] Lakos, C.A.: From Coloured Petri Nets to Object Petri Nets, In ‘16th International Conference on the Application and Theory of Petri Nets’, Nummer 935 in Lecture Notes in Computer Science, Torino, Italy, Springer Verlag, 1995

[Larman 98] Larman, C.: Applying UML and Patterns, Prentice Hall, 1998

[Louden 94] Louden, K. C.: Programmiersprachen : Grundlagen, Konzepte, Entwurf ; 1.Aufl.,Bonn, Internat. Thomson Publ., 1994

[Lyngsø 98] Lyngsø, R.B.; Mailund, T.: Textual Interchange Format for High-Level Petri Nets, University of Aarhus, Computer Science Department, 1998

[Maier 96] Maier, C.: Darstellung von Konzepten der objektorientierten Modellierung und Programmierung mit Petrinetzen, Studienarbeit, Universität Hamburg, Fachbereich Informatik, 1996

[Maier 97] Maier, C.: Objektorientierte Analyse mit gefärbten Petrinetzen, Diplomarbeit, Universität Hamburg, Fachbereich Informatik, 1997

[Maier 98] Maier, C.; Moldt, D.; Rölke, H.: SNIFF: An Input/Output Library for Design/CPN, Universität Hamburg, Fachbereich Informatik, 1998

[Marsan 93] Marsan, M. A.: Application and Theory of Petri Nets 1993, 14th International Conference, Chicago, Illinois, USA, June 1993 Proceedings; Lecture Notes in Computer Science 691 , Springer Verlag, Berlin Heidelberg New York, 1993

[Martin & Odell 95] Martin, J.; Odell, J.: Object-Oriented Methods: A Foundation, Prentice Hall, Englewood Cliffs, NY, 1995

[Matthes 97] Matthes, F.: Mobile Processes in Cooperative Information Systems, In Proceedings STJA `97, Erfurt, Germany, September 1997, Springer Verlag

[McMenamin und Palmer 88] McMenamin, S.; Palmer, J.: Strukturierte Systemanalyse, Coedition von Hanser und Prentice Hall, London, 1988

[Meyer 88] Meyer, B.: Object-Oriented Software Construction, Prentice Hall, New York, 1988

[Meyer 90] Meyer, B.: Objektorientierte Softwareentwicklung; München, Hanser, 1990

[Meyer 98a] Meyer, M.C.: Architekturen zum transparenten Entwurf mit Petrinetzen, Dissertation, Universität Hamburg, Fachbereich Informatik, 1998

[Meyer 98] Meyer, M.C.; Moldt, D.; Netzebandt, M.: Transparenter Entwurf mit gefärbten Petrinetzen am Beispiel eines Reiseunternehmens, Mitteilung 282, Universität Hamburg, Fachbereich Informatik, 1998

[Moldt 96] Moldt, D.: Höhere Petrinetze als Grundlage für Systemspezifikationen, Dissertation, Universität Hamburg, Fachbereich Informatik, 1996

[Netzebandt 97] Netzebandt, M.: Modellierung und Simulation eines Reiseunternehmens mit gefärbten Petrinetzen, Studienarbeit, Universität Hamburg, 1997

[Object Constraint Language Specification] http://www.rational.com/uml/resources/documentation/

[ObjektSpektrum 98] OBJEKTspektrum, 5/98, S.48 + S.49

[Oesterreich 97] Oesterreich, B.: Objektorientierte Softwareentwicklung mit UML, Oldenbourg Verlag, 1997

[Petri 62] Petri, C. A.: Kommunikation mit Automaten, Dissertation, Rheinisch-Westfälisches Institut für Instrumentelle Mathematik an der Universität Bonn, 1962

[Pomberger und Blaschek 93] Pomberger, G.; Blaschek, G.: Grundlagen des Software Engineering: Prototyping und objektorientierte Software-Entwicklung, Hanser Verlag München Wien, 1993

[Quatrani 98] Quatrani, T.: Visual Modeling with Rational Rose and UML, Addison-Wesley, 1997

[Rölke 99] Rölke, H.: Transformation von Klassendiagrammen in objektorientierte Petrinetze unter besonderer Berücksichtigung von Assoziationen, Universität Hamburg, Fachbereich Informatik, Januar 1999

[Rumbaugh et al. 91] Rumbaugh, J.; Blaha, M.; Premeralani, W., Eddy, F.; Lorensen, W.: Object-Oriented Modeling and Design, Prentice-Hall, Englewood Cliffs, New Jersey, 1991

[Scheschonk und Reisig 93] Scheschonk, G.; Reisig, W.: Petrinetze im Einsatz für Entwurf und Entwicklung von Informationssystemen, Informatik Aktuell, Springer Verlag, Berlin, Heidelberg, New York, 1993

[Siegel 95] Siegel, Stefan: Objektorientierter Entwurf mit gefärbten Petrinetzen, Diplomarbeit, Universität Hamburg, Fachbereich Informatik, 1995

[Störrle 98] Störrle, H.: An Evaluation of High-End Tools for Petri-Nets, Bericht 9802, Ludwig-Maximilians-Universität München, Institut für Informatik, 1998

[Texel 97] Texel, P. P.; Williams, C. B.: Use Cases combined with Booch, OMT, UML, Prentice Hall, 1997

[UML 1.1 Extension for Business Modeling] http://www.rational.com/uml/resources/documentation/

[UML 1.1 Extension for Objectory Process for Software Engineering] http://www.rational.com/uml/resources/documentation/

[UML 1.1 Notation-Guide] http://www.rational.com/uml/resources/documentation/

[UML 1.1 Semantics] http://www.rational.com/uml/resources/documentation/

[UML 1.1 Summary] http://www.rational.com/uml/resources/documentation/

[v.d.Aalst 97] van der Aalst, W.M.P.: Verification of Petri Nets, Application and Theory of Petri Nets 1997, 18th international conference, ICATPN´97, Toulouse, France, June 1997

[Wikström 87] Wikström, A.: Functional Programming Using Standard ML, Prentice-Hall, 1987

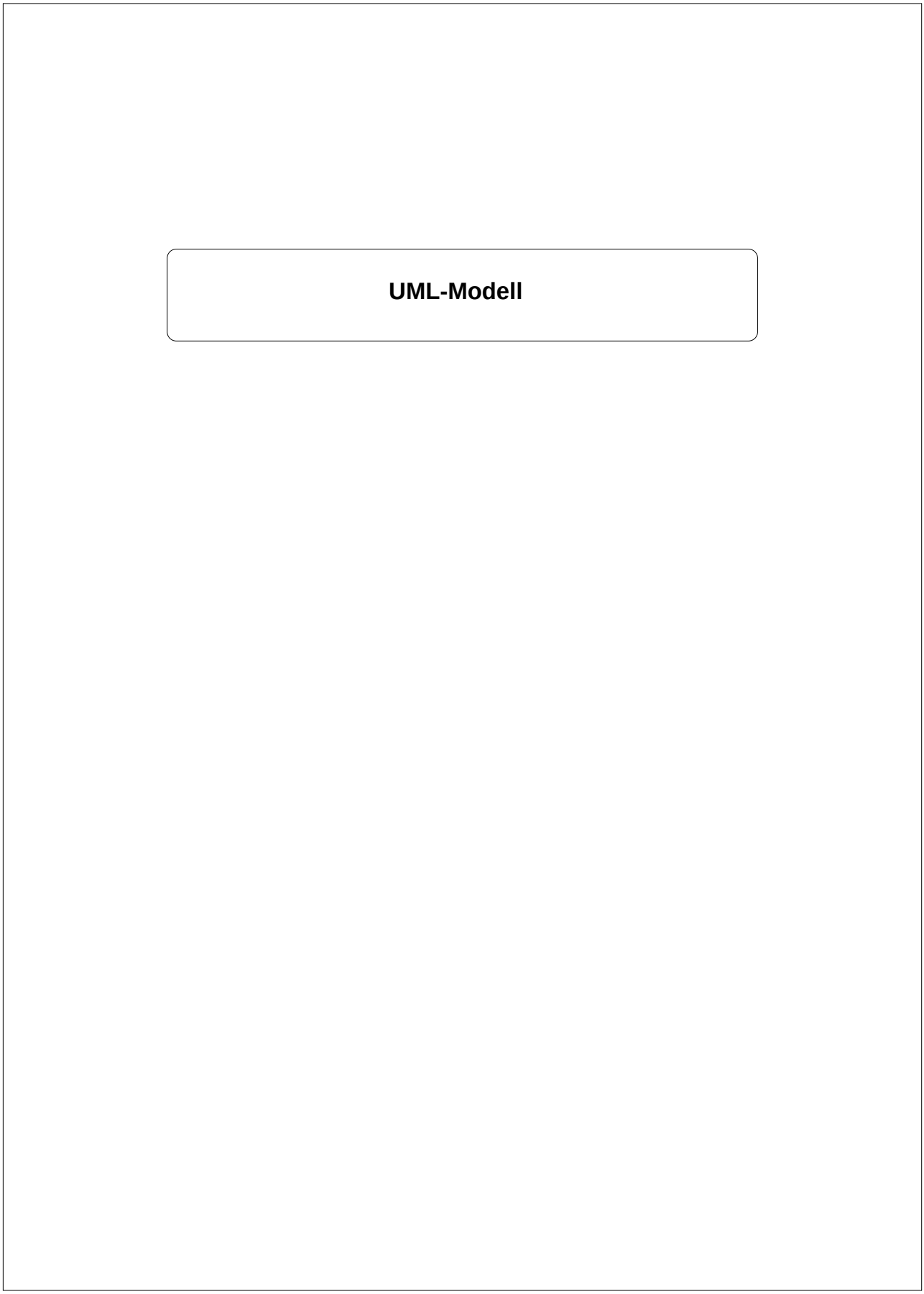
[Wirfs-Brock et al. 90] Wirfs-Brock, R.; Wilkerson, B.; Wiener, L.: Designing Object-Oriented Software, Prentice Hall, Englewood Cliffs, 1990

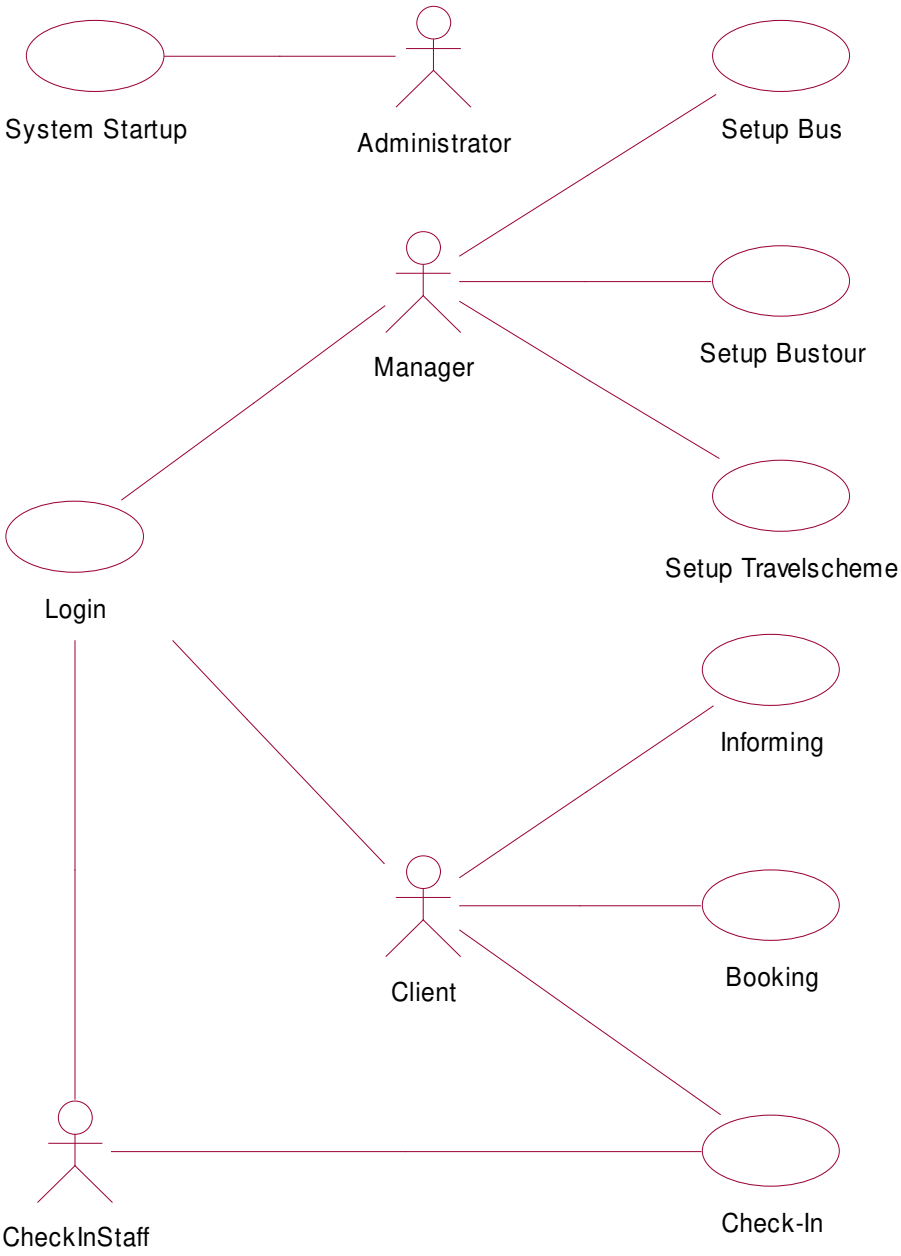
[Wirth 72] Wirth, N.: Systematisches Programmieren - eine Einführung, Teubner, Stuttgart, 1972

[Yourdon 89] Yourdon, E.: Modern Structured Analysis, Prentice-Hall, Englewood Cliffs New Jersey, 1989

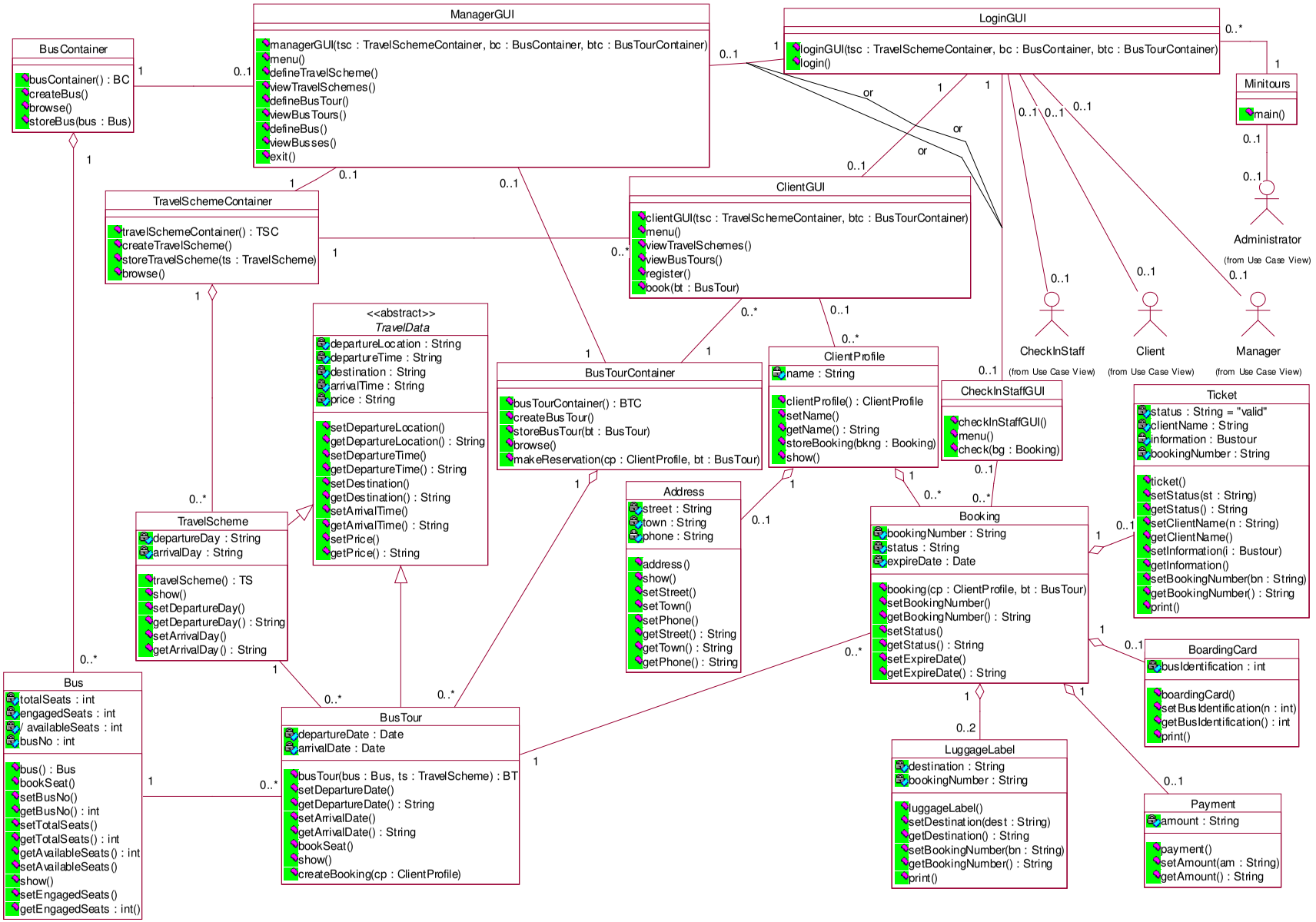
8 ANHANG

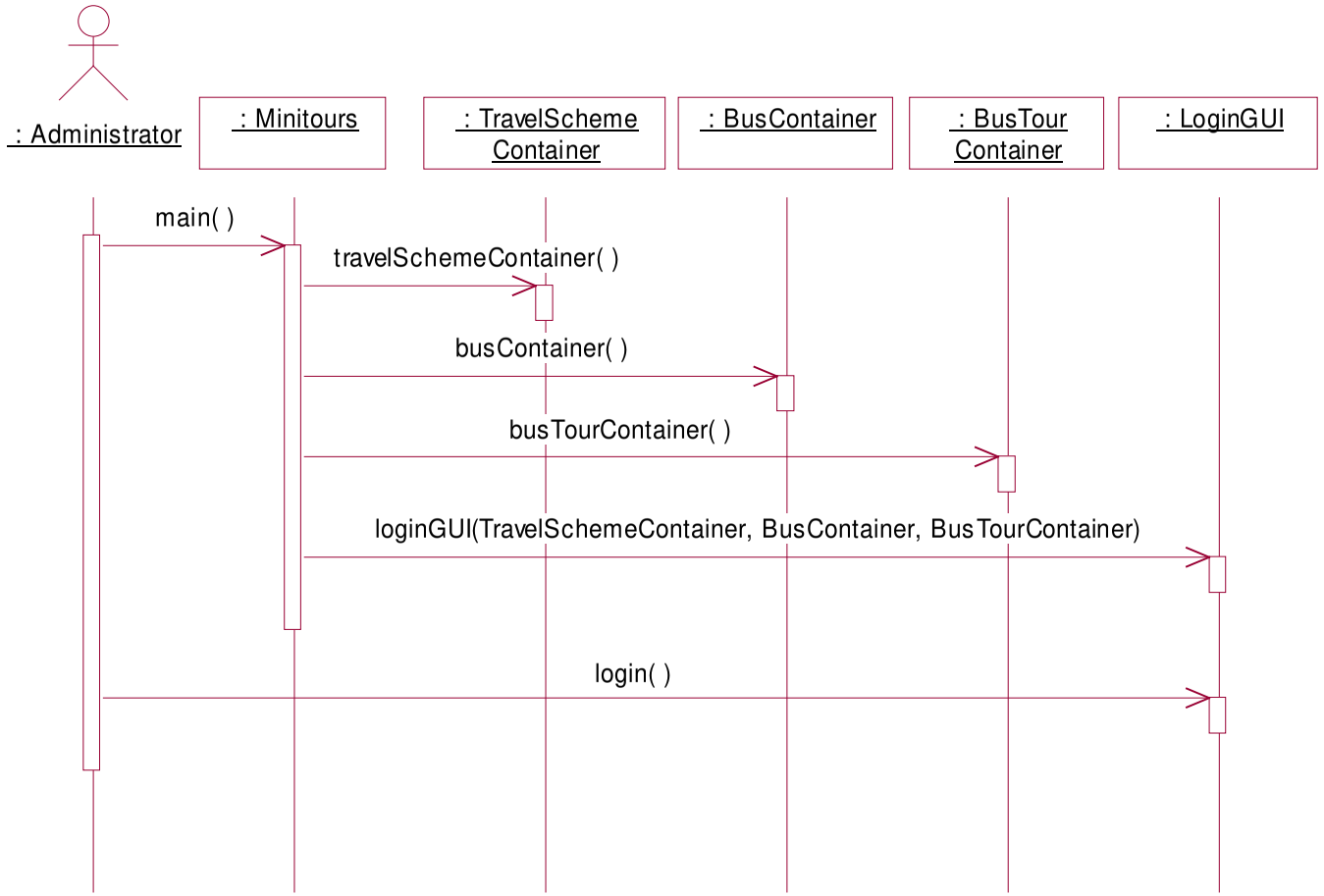
In diesem Anhang sind die Diagramme in der UML-Notation in etwas größerer Darstellung als im Text abgebildet. Der zweiten Teil enthält alle Netzteile der ausführbaren Systemspezifikation des Reiseunternehmens Minitours. Die Abbildungen im letzten Teil dieses Anhangs zeigt zusammenfassend die im Fallbeispiel verwendeten Notationen. Die Abbildungen werden an dieser Stelle nicht kommentiert, weil die Notation und deren Verwendung im Laufe dieser Arbeit ausführlich erläutert wird.


UML-Modell

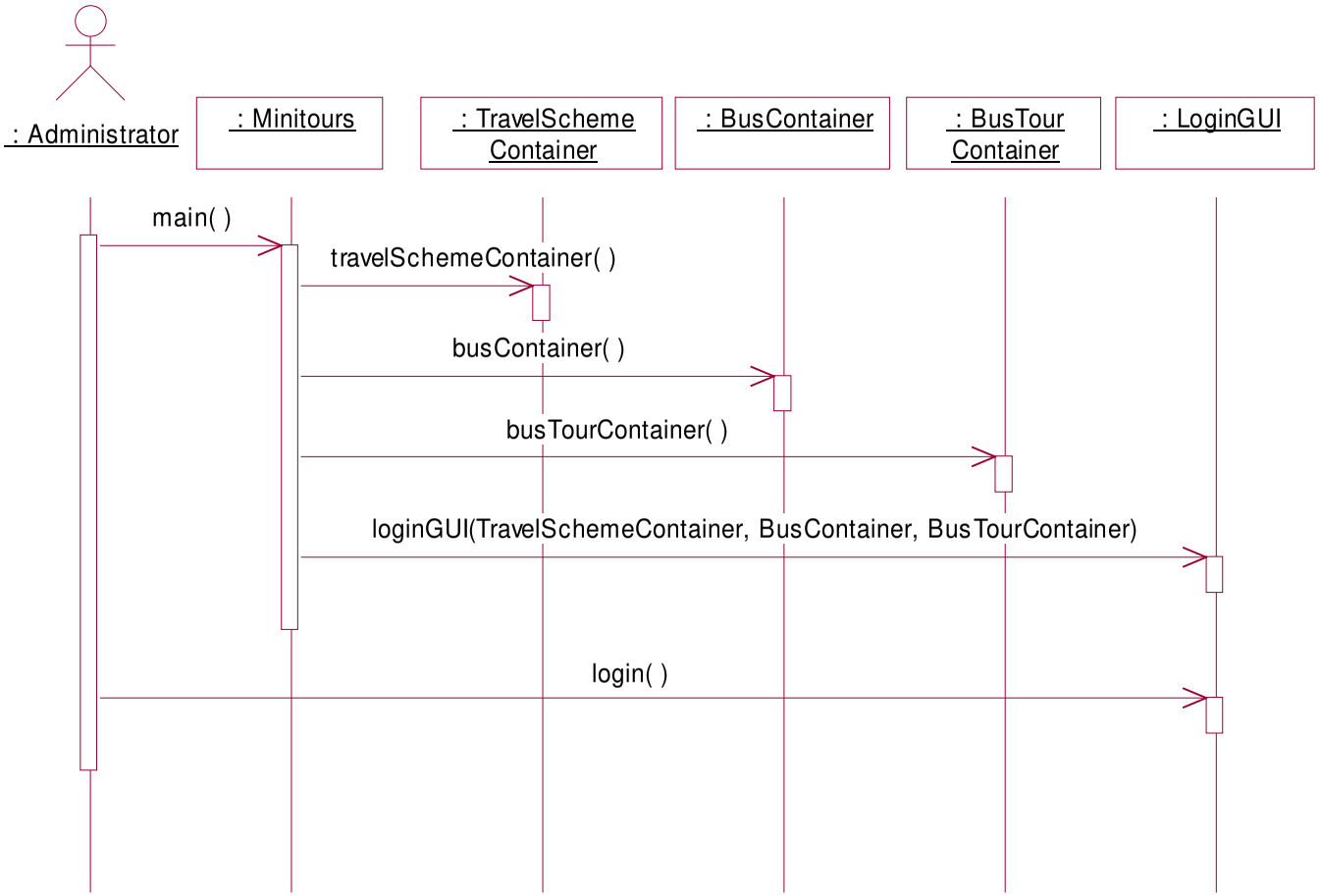


Use case diagram

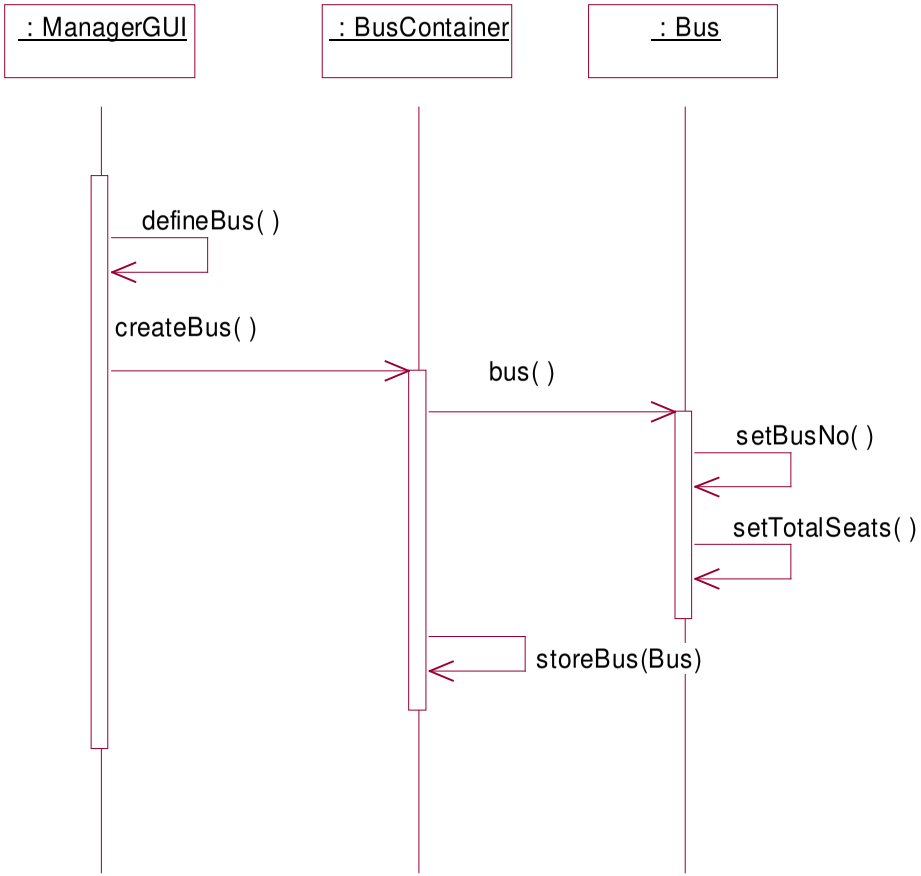

ManagerGUI
managerGUI(tsc : TravelSchemeContainer, bc : BusContainer, btc : BusTourContainer)
menu()
defineTravelScheme()
viewTravelSchemes()
defineBusTour()
viewBusTours()
defineBus()
viewBusses()
exit()
LoginGUI
loginGUI(tsc : TravelSchemeContainer, bc : BusContainer, btc : BusTourContainer)
login()
BusContainer
busContainer() : BC
createBus()
browse()
storeBus(bus : Bus)
Minitours
main()
Administrator
(from Use Case View)
TravelSchemeContainer
travelSchemeContainer() : TSC
createTravelScheme()
storeTravelScheme(ts : TravelScheme)
browse()
ClientGUI
clientGUI(tsc : TravelSchemeContainer, btc : BusTourContainer)
menu()
viewTravelSchemes()
viewBusTours()
register()
book(bt : BusTour)
CheckInStaff
(from Use Case View)
Client
(from Use Case View)
Manager
(from Use Case View)
<<abstract>>
TravelData
departureLocation : String
departureTime : String
destination : String
arrivalTime : String
price : String
setDepartureLocation()
getDepartureLocation() : String
setDepartureTime()
getDepartureTime() : String
setDestination()
getDestination() : String
setArrivalTime()
getArrivalTime() : String
setPrice()
getPrice() : String
BusTourContainer
busTourContainer() : BTC
createBusTour()
storeBusTour(bt : BusTour)
browse()
makeReservation(cp : ClientProfile, bt : BusTour)
ClientProfile
name : String
clientProfile() : ClientProfile
setName()
getName() : String
storeBooking(bkng : Booking)
show()
CheckInStaffGUI
checkInStaffGUI()
menu()
check(bg : Booking)
Ticket
status : String = "valid"
clientName : String
information : Bustour
bookingNumber : String
ticket()
setStatus(st : String)
getStatus() : String
setClientName(n : String)
getClientName()
setInformation(i : Bustour)
getInformation()
setBookingNumber(bn : String)
getBookingNumber() : String
print()
TravelScheme
departureDay : String
arrivalDay : String
travelScheme() : TS
show()
setDepartureDay()
getDepartureDay() : String
setArrivalDay()
getArrivalDay() : String
Address
street : String
town : String
phone : String
address()
show()
setStreet()
setTown()
setPhone()
getStreet() : String
getTown() : String
getPhone() : String
Booking
bookingNumber : String
status : String
expireDate : Date
booking(cp : ClientProfile, bt : BusTour)
setBookingNumber()
getBookingNumber() : String
setStatus()
getStatus() : String
setExpireDate()
getExpireDate() : String
BoardingCard
busIdentification : int
boardingCard()
setBusIdentification(n : int)
getBusIdentification() : int
print()
Bus
totalSeats : int
engagedSeats : int
/ availableSeats : int
busNo : int
bus() : Bus
bookSeat()
setBusNo()
getBusNo() : int
setTotalSeats()
getTotalSeats() : int
getAvailableSeats() : int
setAvailableSeats()
show()
setEngagedSeats()
getEngagedSeats : int()
BusTour
departureDate : Date
arrivalDate : Date
busTour(bus : Bus, ts : TravelScheme) : BT
setDepartureDate()
getDepartureDate() : String
setArrivalDate()
getArrivalDate() : String
bookSeat()
show()
createBooking(cp : ClientProfile)
LuggageLabel
destination : String
bookingNumber : String
luggageLabel()
setDestination(dest : String)
getDestination() : String
setBookingNumber(bn : String)
getBookingNumber() : String
print()
Payment
amount : String
payment()
setAmount(am : String)
getAmount() : String



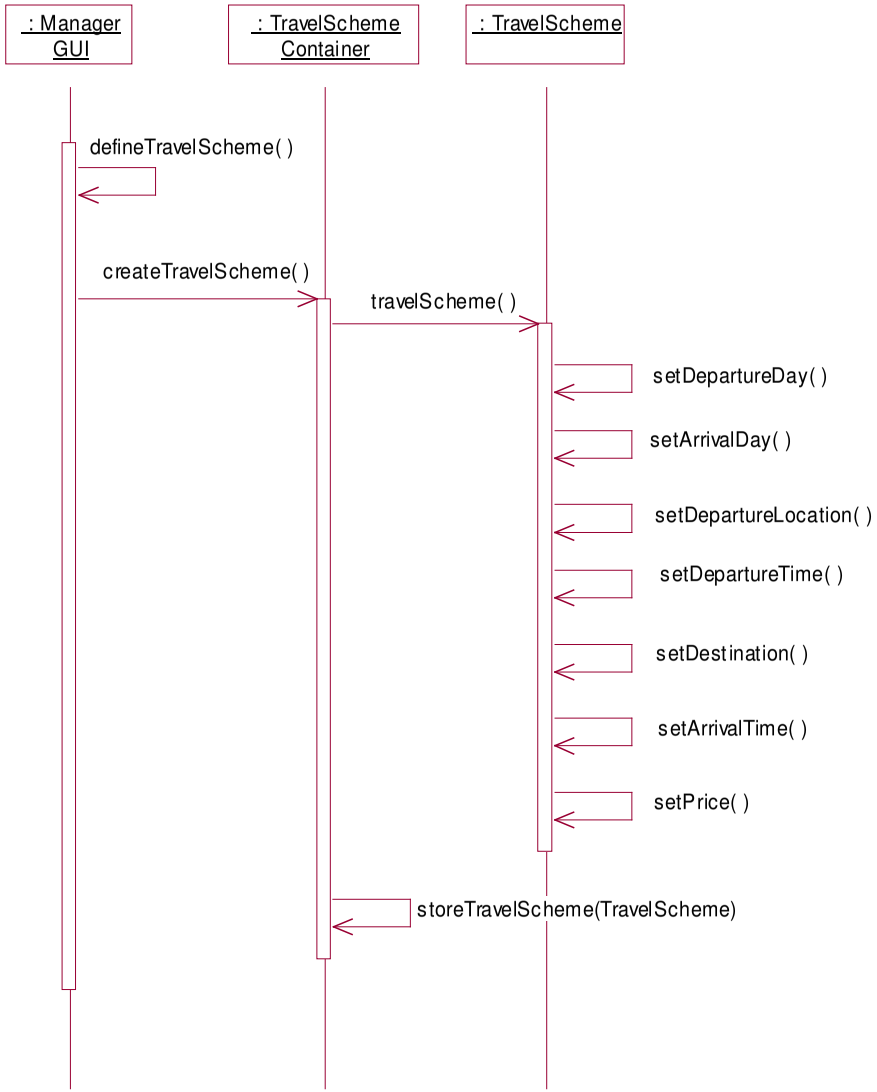
Startup of the system



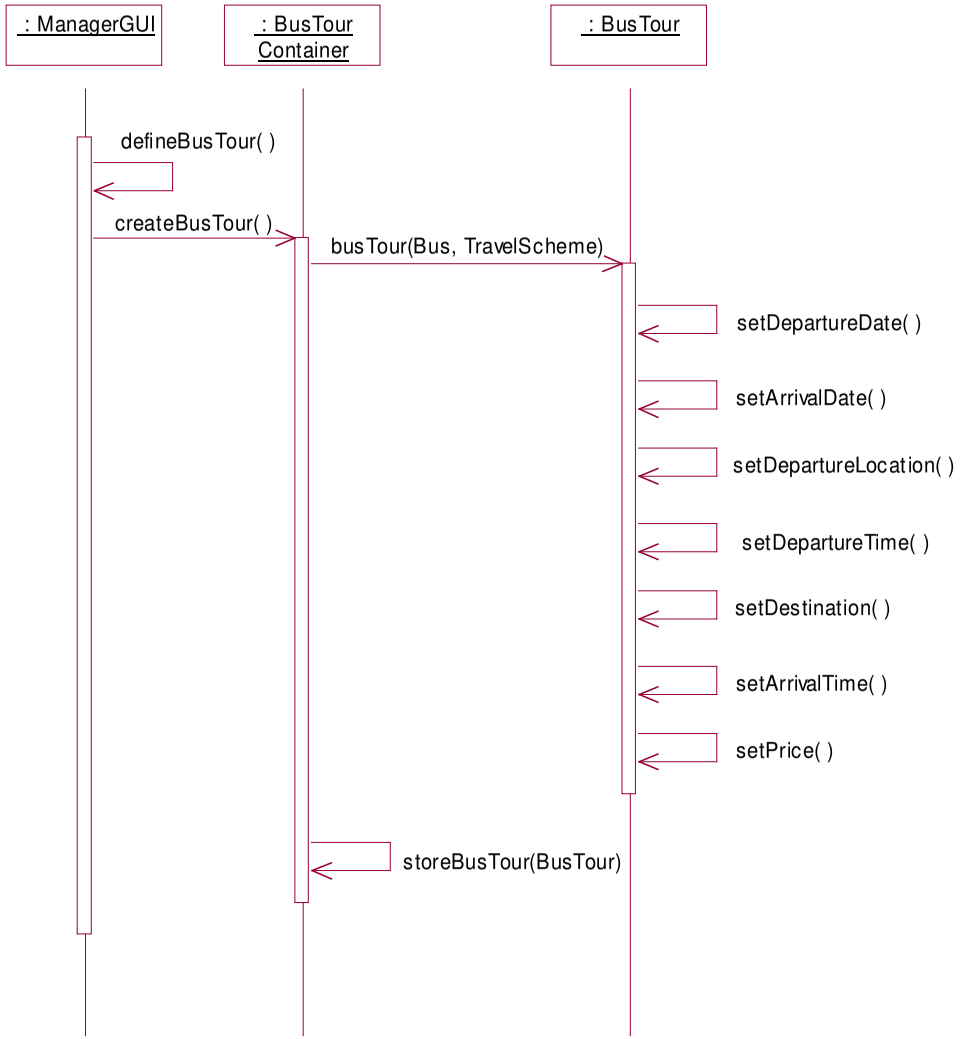
Startup of the system



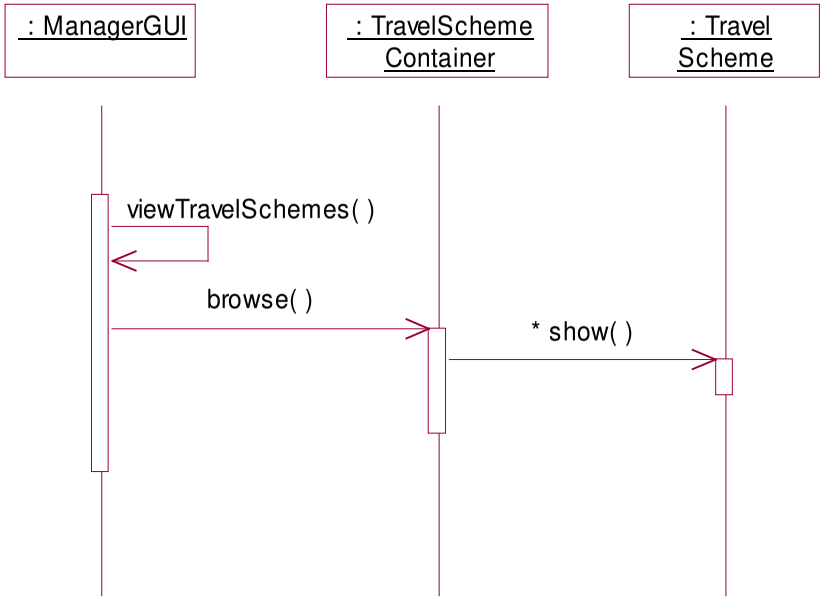
Definition of a bus



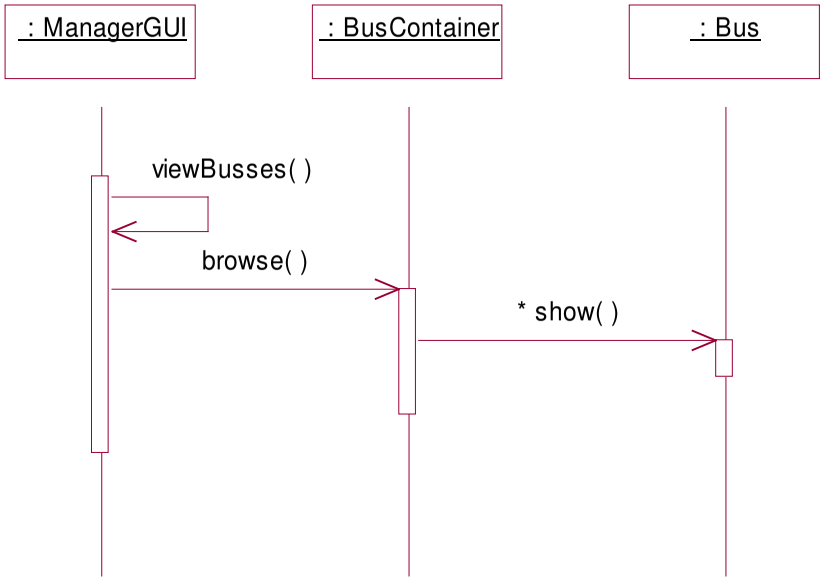
Definition of a travel scheme



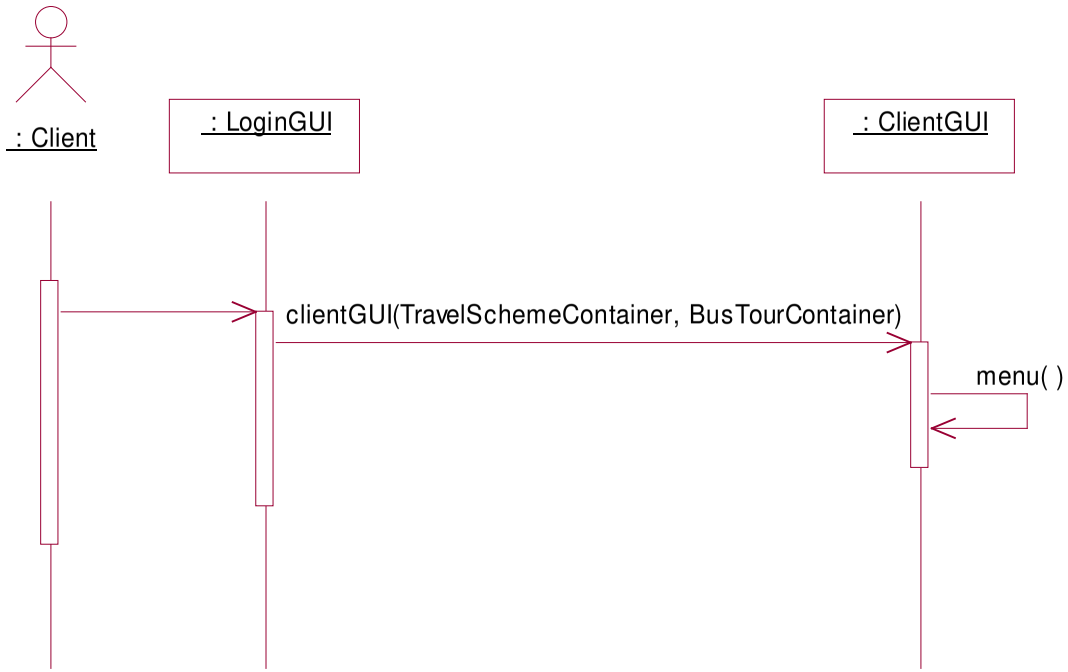
Definition of a bus tour



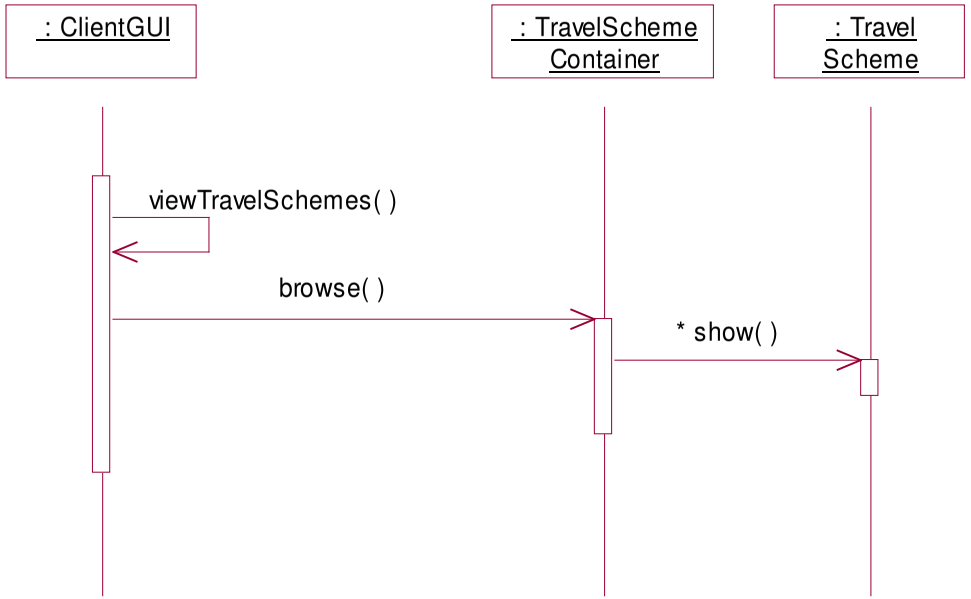
Browsing of the travel schemes



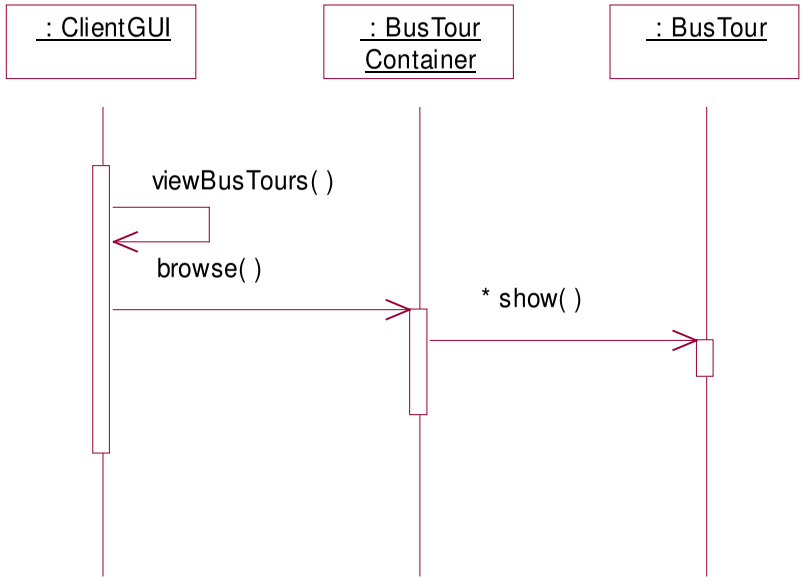
Browsing of the busses



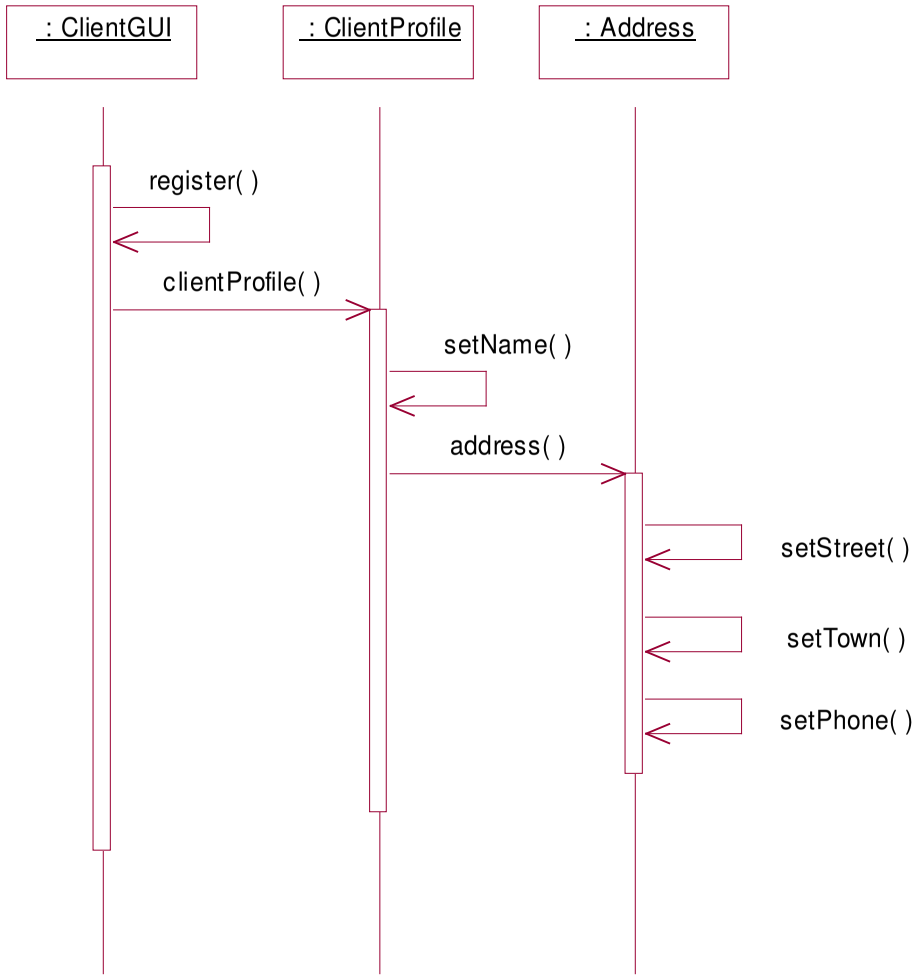
Login of a client



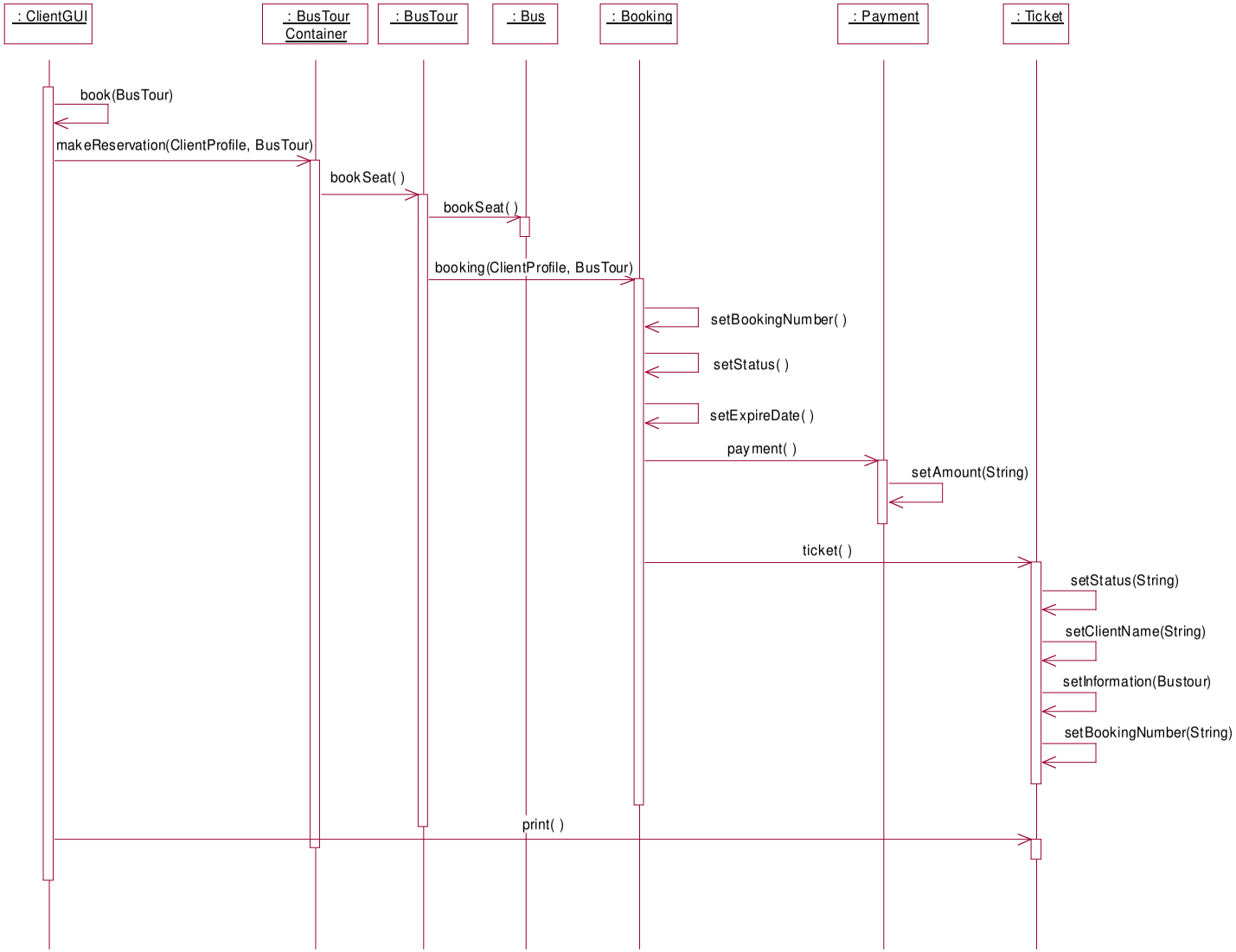
Browsing of the travel schemes



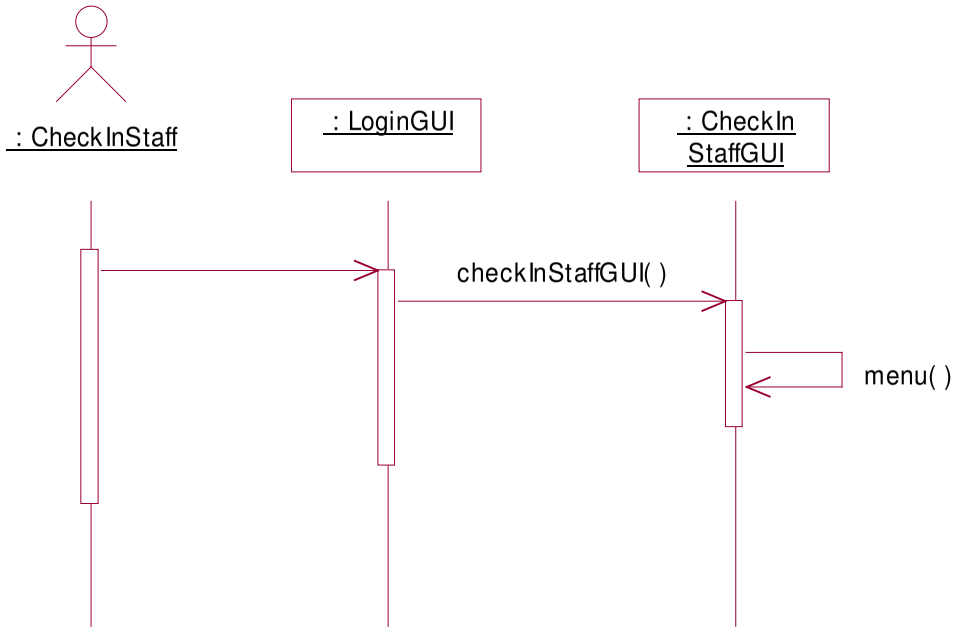
Browsing of the bus tours



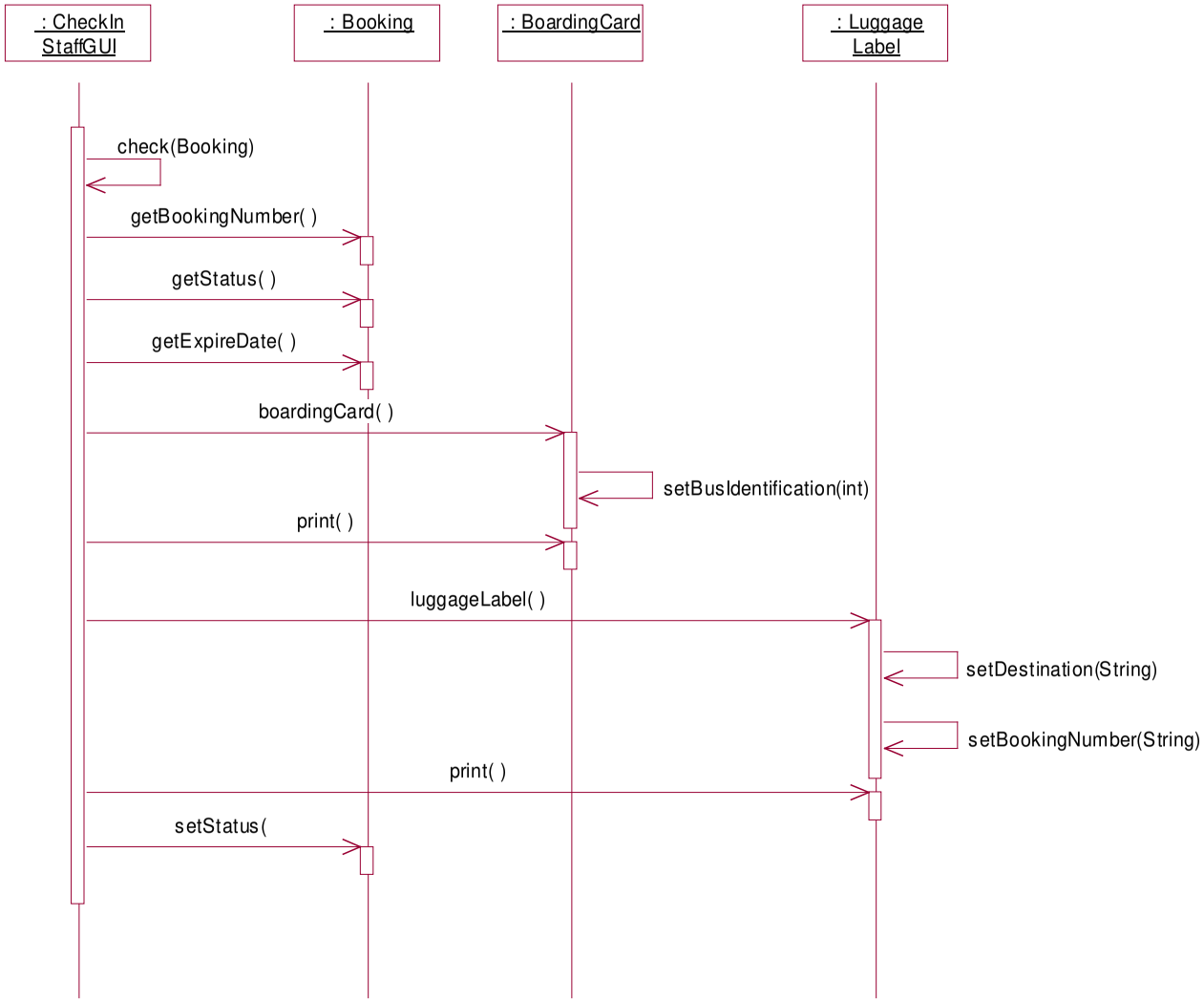
Registration of a client



Booking of a bus tour

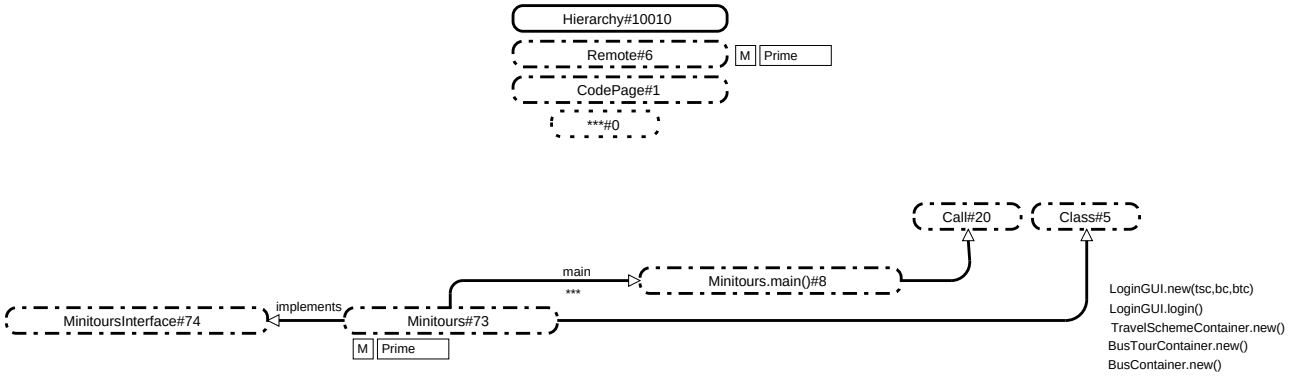


Login of the check-in staff

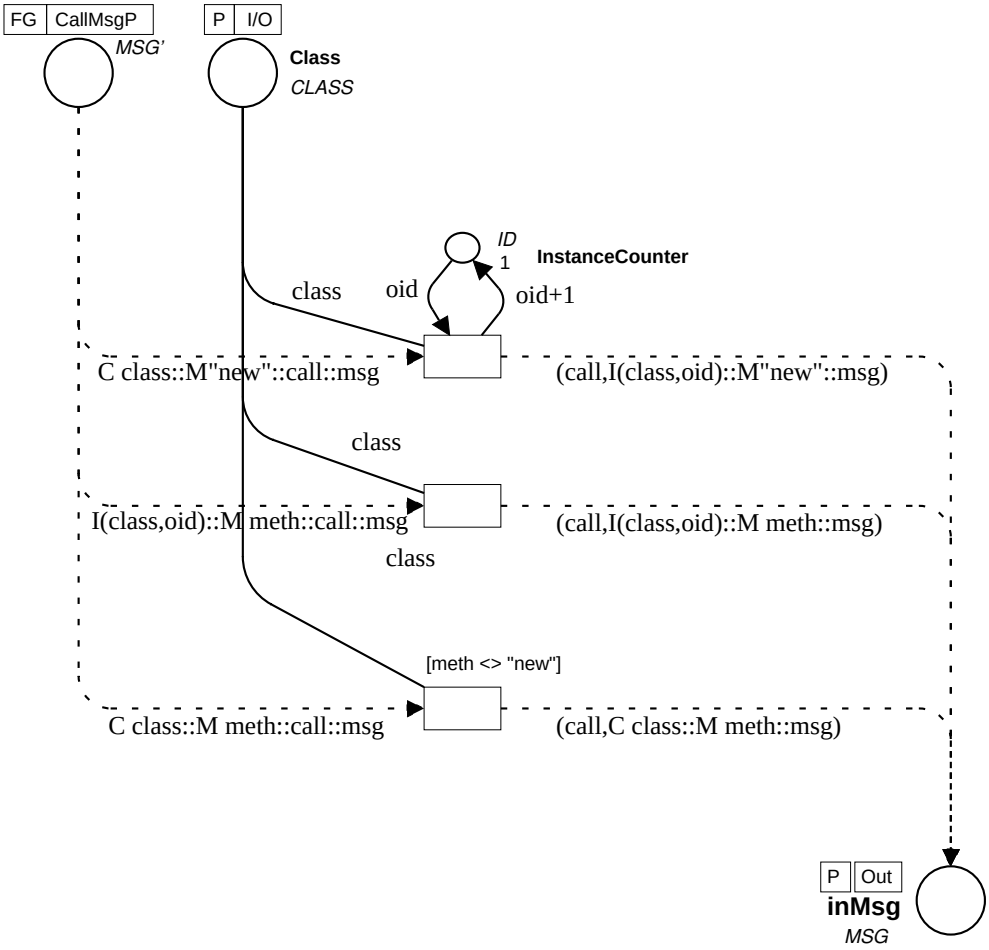


Check-in of a client

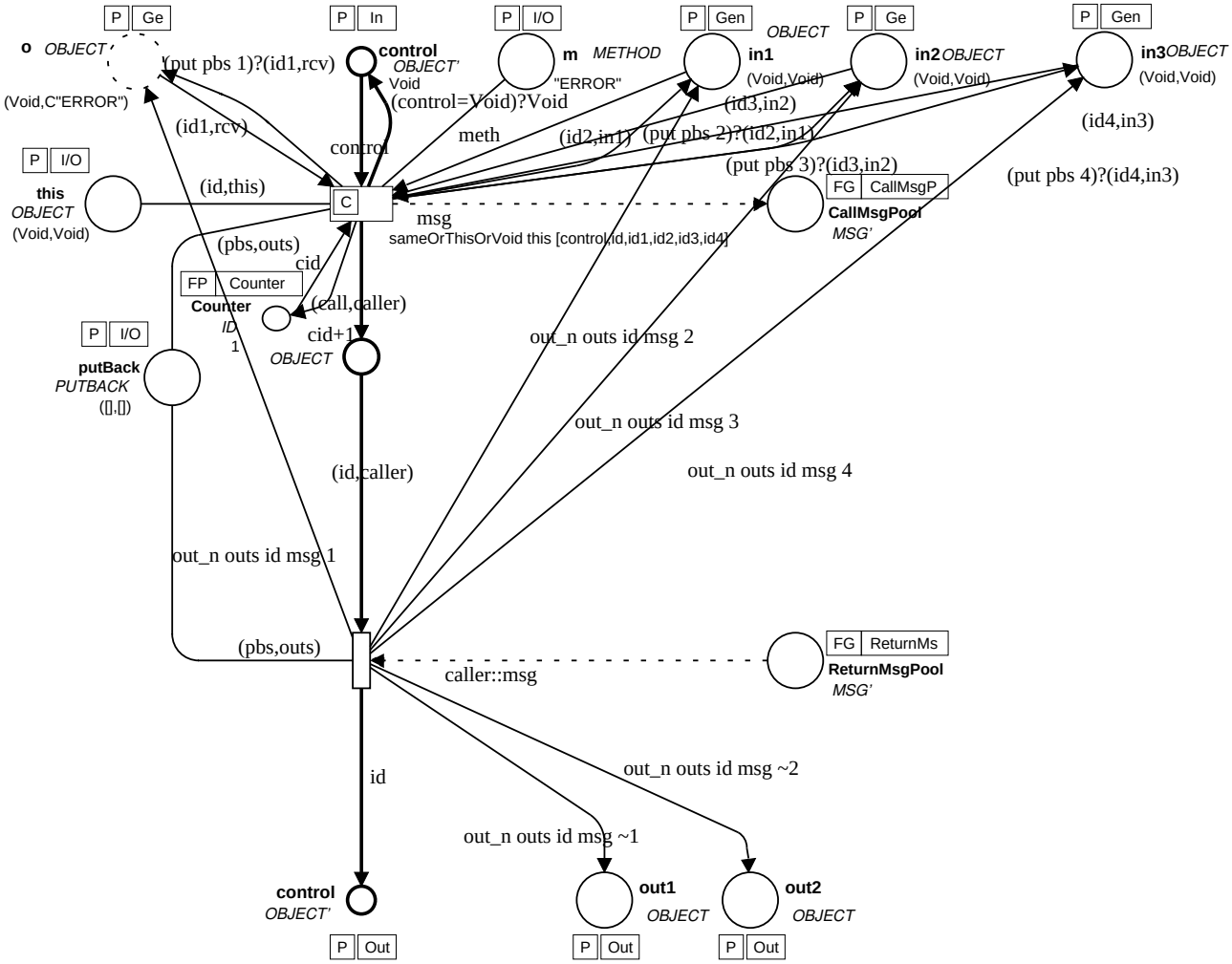
Petrinetz-Modell


Hierarchy#10010
Remote#6
M
Prime
CodePage#1
***#0
Call#20
Class#5
main
***
Minitours.main()#8
MinitoursInterface#74
implements
Minitours#73
M
Prime
LoginGUI.new(tsc,bc,btc)
LoginGUI.login()
TravelSchemeContainer.new()
BusTourContainer.new()
BusContainer.new()

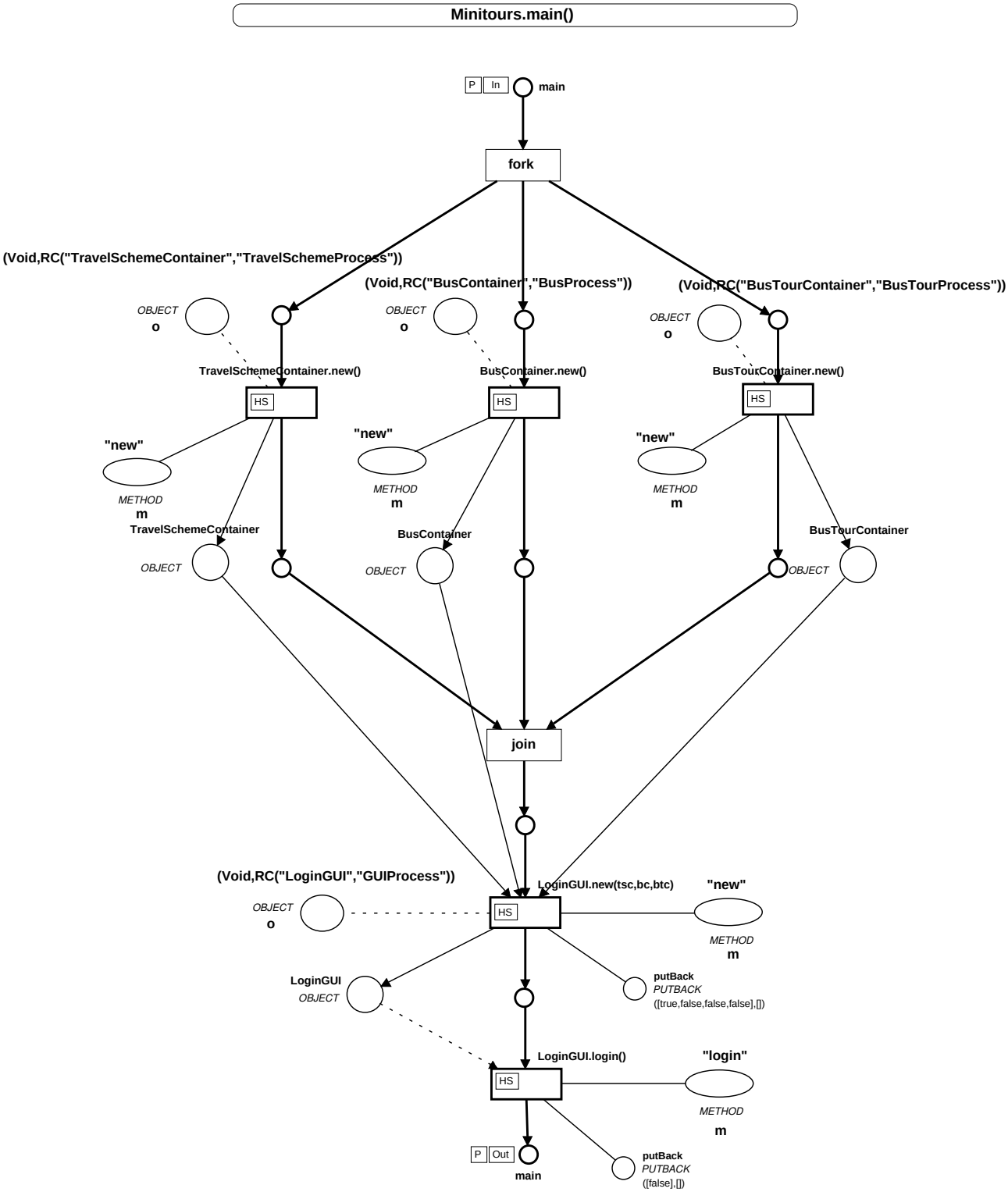
Class-Page


FG
CallMsgP
MSG'
P
I/O
Class
CLASS
ID
1
InstanceCounter
class
oid
oid+1
C class::M"new"::call::msg
(call,I(class,oid)::M"new"::msg)
class
I(class,oid)::M meth::call::msg
(call,I(class,oid)::M meth::msg)
class
[meth <> "new"]
C class::M meth::call::msg
(call,C class::M meth::msg)
P
Out
inMsg
MSG

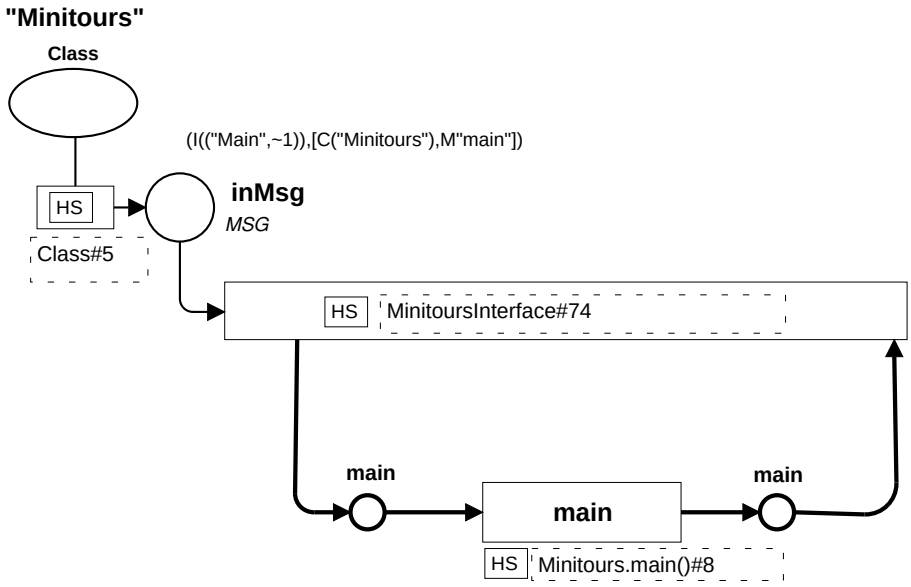
Call-Page

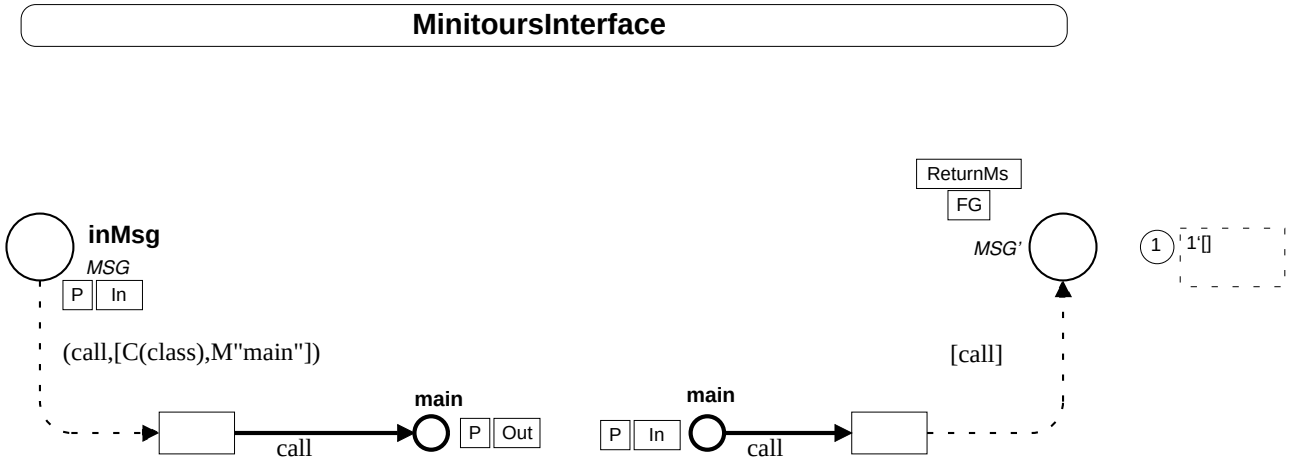


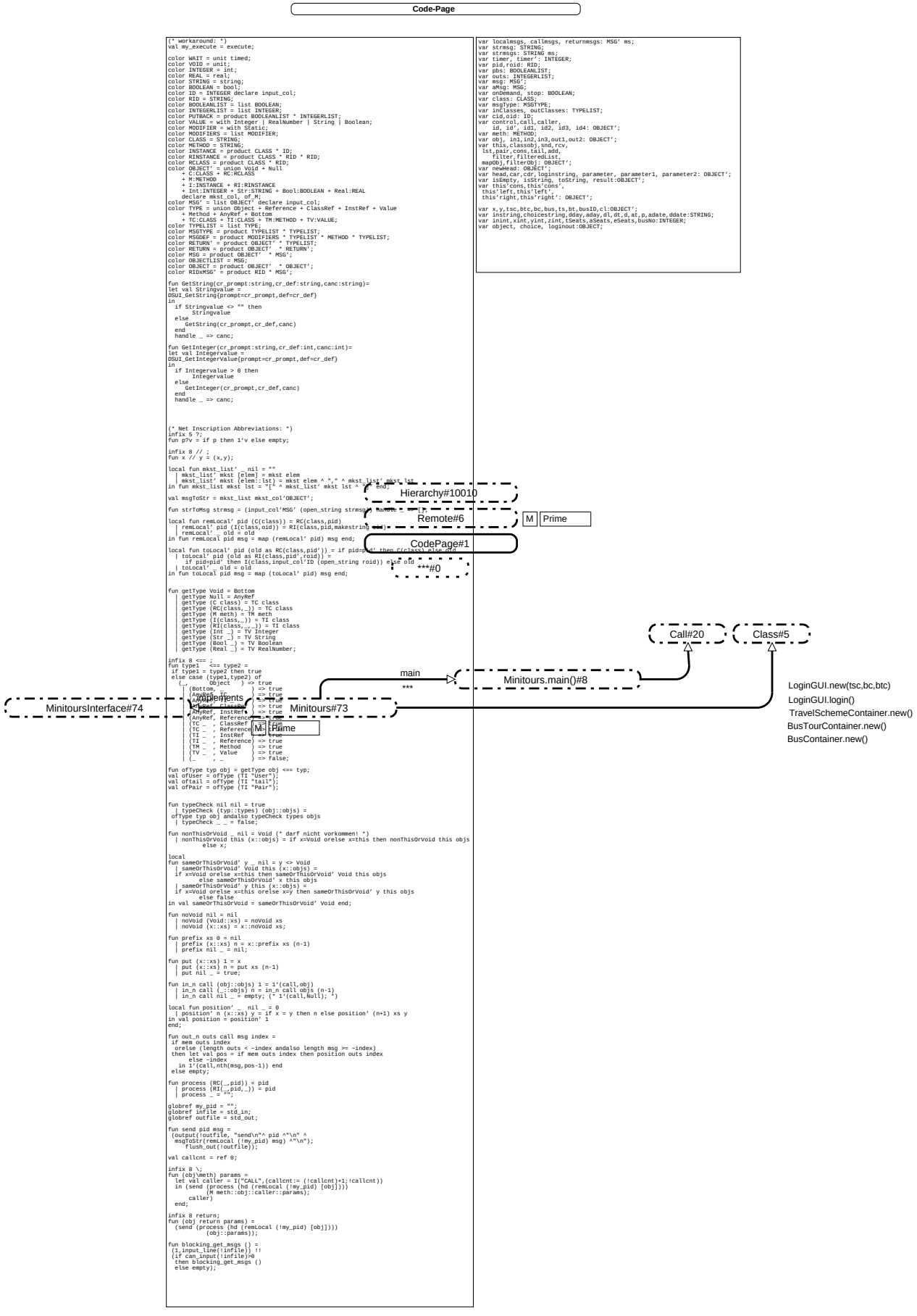

```
input (control,id,this,id1,rcv,id2,meth,id3,in1,in2,cid,in3,id4);
output(call,caller,msg);
action
  let val cntrl = nonThisOrVoid this [control,id,id1,id2,id3,id4];
    val caller = I("Call",cid);
in (cntrl,caller,rcv::M meth::caller::(noVoid [in1,in2,in3]))
end;
```


Minitours.main()
P
In
main
fork
(Void,RC("TravelSchemeContainer","TravelSchemeProcess"))
(Void,RC("BusContainer","BusProcess"))
(Void,RC("BusTourContainer","BusTourProcess"))
OBJECT
o
TravelSchemeContainer.new()
BusContainer.new()
BusTourContainer.new()
HS
"new"
METHOD
m
TravelSchemeContainer
BusContainer
BusTourContainer
join
(Void,RC("LoginGUI","GUIProcess"))
LoginGUI.new(tsc,bc,btc)
LoginGUI
putBack
PUTBACK
([true,false,false,false],[])
LoginGUI.login()
"login"
Out
([false],[])

Minitours

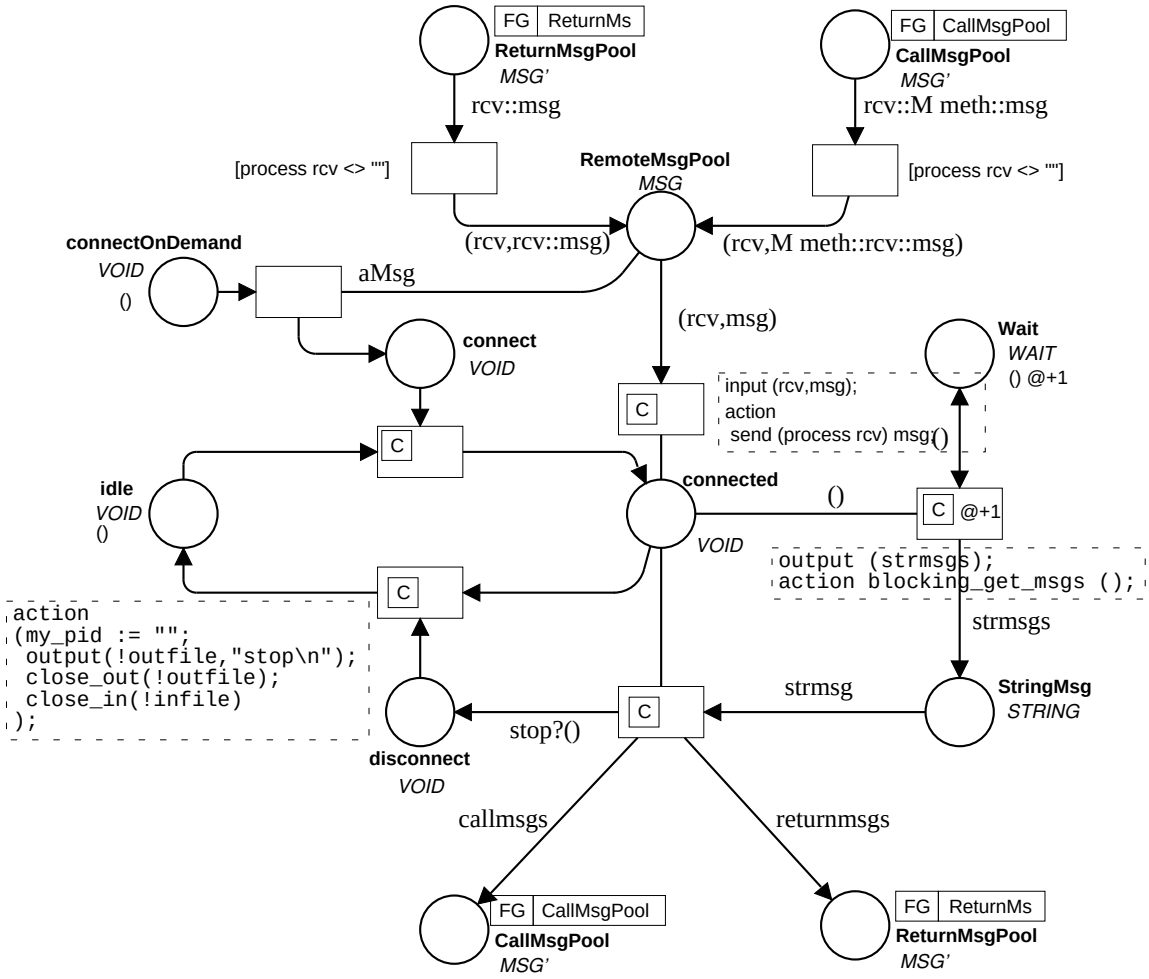

"Minitours"
Class
(I(("Main",~1)),[C("Minitours"),M"main"])
HS
inMsg
MSG
Class#5
HS
MinitoursInterface#74
main
main
main
HS
Minitours.main()#8


MinitoursInterface
inMsg
MSG
P
In
(call,[C(class),M"main"])
call
main
P
Out
main
P
In
call
[call]
ReturnMs
FG
MSG'
1
1`[]

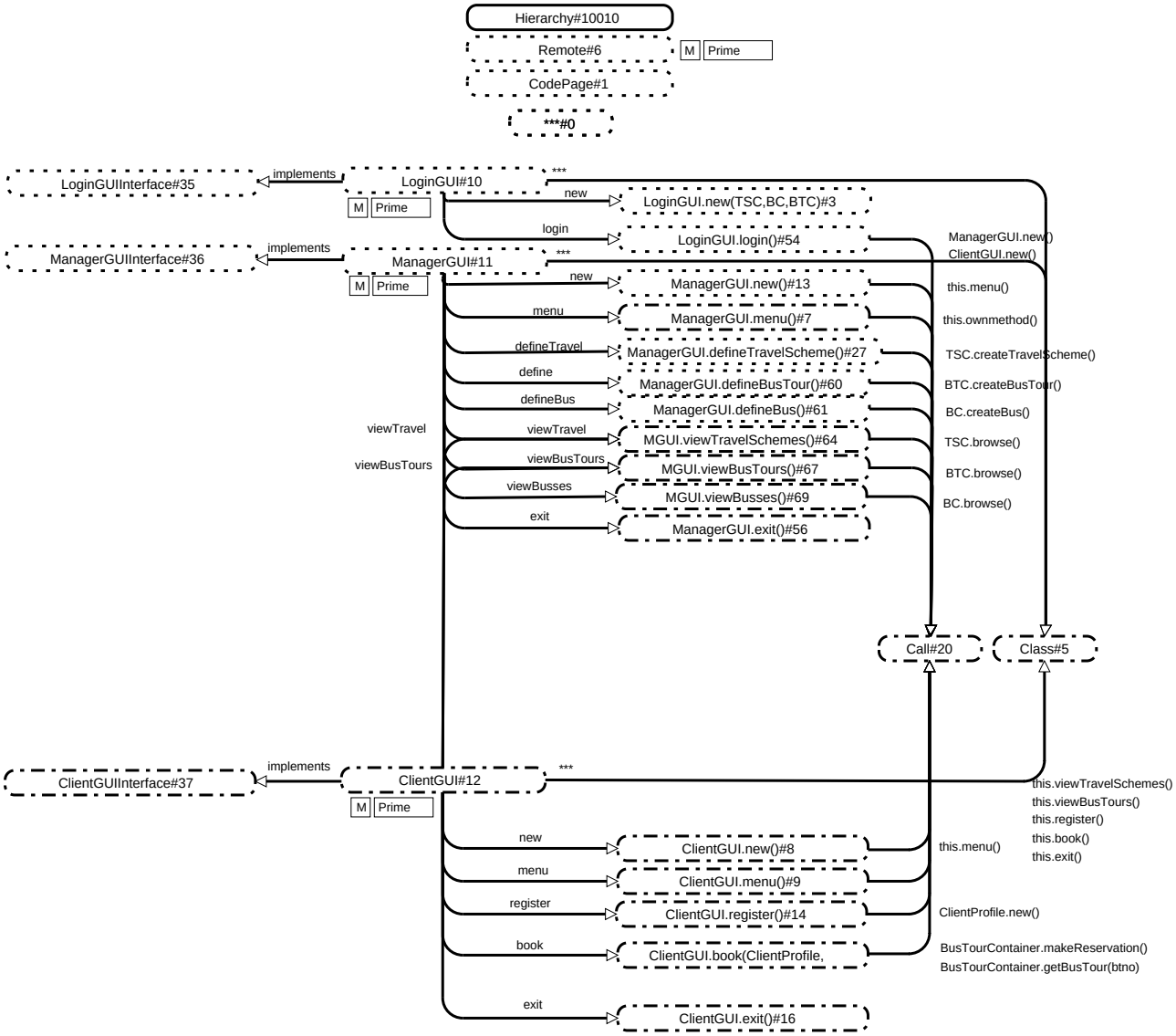

Code-Page
(* workaround: *)
val my_execute = execute;
color WAIT = unit timed;
color VOID = unit;
color INTEGER = int;
color REAL = real;
color STRING = string;
color BOOLEAN = bool;
color ID = INTEGER declare input_col;
color RID = STRING;
color BOOLEANLIST = list BOOLEAN;
color INTEGERLIST = list INTEGER;
color PUTBACK = product BOOLEANLIST * INTEGERLIST;
color VALUE = with Integer | RealNumber | String | Boolean;
color MODIFIER = with Static;
color MODIFIERS = list MODIFIER;
color CLASS = STRING;
color METHOD = STRING;
color INSTANCE = product CLASS * ID;
color RINSTANCE = product CLASS * RID * RID;
color RCLASS = product CLASS * RID;
color OBJECT' = union Void + Null
+ C:CLASS + RC:RCLASS
+ M:METHOD
+ I:INSTANCE + RI:RINSTANCE
+ Int:INTEGER + Str:STRING + Bool:BOOLEAN + Real:REAL
declare mkst_col, of_M;
color MSG' = list OBJECT' declare input_col;
color TYPE = union Object + Reference + ClassRef + InstRef + Value
+ Method + AnyRef + Bottom
+ TC:CLASS + TI:CLASS + TM:METHOD + TV:VALUE;
color TYPELIST = list TYPE;
color MSGTYPE = product TYPELIST * TYPELIST;
color MSGDEF = product MODIFIERS * TYPELIST * METHOD * TYPELIST;
color RETURN' = product OBJECT' * TYPELIST;
color RETURN = product OBJECT' * RETURN';
color MSG = product OBJECT' * MSG';
color OBJECTLIST = MSG;
color OBJECT = product OBJECT' * OBJECT';
color RIDxMSG' = product RID * MSG';
fun GetString(cr_prompt:string,cr_def:string,canc:string)=
let val Stringvalue =
DSUI_GetString{prompt=cr_prompt,def=cr_def}
in
if Stringvalue <> "" then
Stringvalue
else
GetString(cr_prompt,cr_def,canc)
end
handle _ => canc;
fun GetInteger(cr_prompt:string,cr_def:int,canc:int)=
let val Integervalue =
DSUI_GetIntegerValue{prompt=cr_prompt,def=cr_def}
in
if Integervalue > 0 then
Integervalue
else
GetInteger(cr_prompt,cr_def,canc)
end
handle _ => canc;
(* Net Inscription Abbreviations: *)
infix 5 ?;
fun p?v = if p then 1'v else empty;
infix 8 // ;
fun x // y = (x,y);
local fun mkst_list' _ nil = ""
| mkst_list' mkst [elem] = mkst elem
| mkst_list' mkst (elem::lst) = mkst elem ^ "," ^ mkst_list' mkst lst
in fun mkst_list mkst lst = "[" ^ mkst_list' mkst lst ^ "]" end;
val msgToStr = mkst_list mkst_col'OBJECT';
fun strToMsg strmsg = (input_col'MSG' (open_string strmsg) handle _ => nil);
local fun remLocal' pid (C(class)) = RC(class,pid)
| remLocal' pid (I(class,oid)) = RI(class,pid,makestring oid)
| remLocal' _ old = old
in fun remLocal pid msg = map (remLocal' pid) msg end;
local fun toLocal' pid (old as RC(class,pid')) = if pid=pid' then C(class) else old
| toLocal' pid (old as RI(class,pid',roid)) =
if pid=pid' then I(class,input_col'ID (open_string roid)) else old
| toLocal' _ old = old
in fun toLocal pid msg = map (toLocal' pid) msg end;
fun getType Void = Bottom
| getType Null = AnyRef
| getType (C class) = TC class
| getType (RC(class,_)) = TC class
| getType (M meth) = TM meth
| getType (I(class,_)) = TI class
| getType (RI(class,_,_)) = TI class
| getType (Int _) = TV Integer
| getType (Str _) = TV String
| getType (Bool _) = TV Boolean
| getType (Real _) = TV RealNumber;
infix 8 <== ;
fun type1 <== type2 =
if type1 = type2 then true
else case (type1,type2) of
(_, Object) => true
| (Bottom,_) => true
| (AnyRef,_) => true
| (ClassRef,ClassRef) => true
| (InstRef,InstRef) => true
| (AnyRef,Reference) => true
| (TC _ , ClassRef) => true
| (TC _ , Reference) => true
| (TI _ , InstRef) => true
| (TI _ , Reference) => true
| (TM _ , Method) => true
| (TV _ , Value) => true
| (_ , _) => false;
fun ofType typ obj = getType obj <== typ;
val ofUser = ofType (TI "User");
val ofTail = ofType (TI "Tail");
val ofPair = ofType (TI "Pair");
fun typeCheck nil nil = true
| typeCheck (typ::types) (obj::objs) =
ofType typ obj andalso typeCheck types objs
| typeCheck _ _ = false;
fun nonThisOrVoid _ nil = Void (* darf nicht vorkommen! *)
| nonThisOrVoid this (x::objs) = if x=Void orelse x=this then nonThisOrVoid this objs
else x;
local
fun sameOrThisOrVoid' y _ nil = y <> Void
| sameOrThisOrVoid' Void this (x::objs) =
if x=Void orelse x=this then sameOrThisOrVoid' Void this objs
else sameOrThisOrVoid' x this objs
| sameOrThisOrVoid' y this (x::objs) =
if x=Void orelse x=this orelse x<y then sameOrThisOrVoid' y this objs
else false
in val sameOrThisOrVoid = sameOrThisOrVoid' Void end;
fun noVoid nil = nil
| noVoid (Void::xs) = noVoid xs
| noVoid (x::xs) = x::noVoid xs;
fun prefix xs 0 = nil
| prefix (x::xs) n = x::prefix xs (n-1)
| prefix nil _ = nil;
fun put (x::xs) 1 = x
| put (x::xs) n = put xs (n-1)
| put nil _ = true;
fun in_n call (obj::objs) 1 = 1'(call,obj)
| in_n call (_::objs) n = in_n call objs (n-1)
| in_n call nil _ = empty; (* 1'(call,Null); *)
local fun position' _ nil _ = 0
| position' n (x::xs) y = if x = y then n else position' (n+1) xs y
in val position = position' 1
end;
fun out_n outs call msg index =
if mem outs index
orelse (length outs < ~index andalso length msg >= ~index)
then let val pos = if mem outs index then position outs index
else ~index
in 1'(call,nth(msg,pos-1)) end
else empty;
fun process (RC(_,pid)) = pid
| process (RI(_,pid,_)) = pid
| process _ = "";
globref my_pid = "";
globref infile = std_in;
globref outfile = std_out;
fun send pid msg =
(output(!outfile, "send\n"^ pid ^"\n" ^
msgToStr(remLocal (!my_pid) msg) ^"\n");
flush_out(!outfile));
val callcnt = ref 0;
infix 8 \;
fun (obj)\meth) params =
let val caller = I("CALL",(callcnt:= (!callcnt)+1;!callcnt))
in (send (process (hd (remLocal (!my_pid) [obj])))
(M meth::obj::caller::params);
caller)
end;
infix 8 return;
fun (obj return params) =
(send (process (hd (remLocal (!my_pid) [obj])))
(obj::params));
fun blocking_get_msgs () =
(1,input_line(!infile)) !!
(if can_input(!infile)>0
then blocking_get_msgs ()
else empty);
var localmsgs, callmsgs, returnmsgs: MSG' ms;
var strmsg: STRING;
var strmsgs: STRING ms;
var timer, timer': INTEGER;
var pid,roid: RID;
var pbs: BOOLEANLIST;
var outs: INTEGERLIST;
var msg: MSG';
var aMsg: MSG;
var onDemand, stop: BOOLEAN;
var class: CLASS;
var msgType: MSGTYPE;
var inClasses, outClasses: TYPELIST;
var cid,oid: ID;
var control,call,caller,
id, id', id1, id2, id3, id4: OBJECT';
var meth: METHOD;
var obj, in1,in2,in3,out1,out2: OBJECT';
var this,classobj,snd,rcv,
lst,pair,cons,tail,add,
filter,filteredList,
mapObj,filterObj: OBJECT';
var newHead: OBJECT';
var head,car,cdr,loginstring, parameter, parameter1, parameter2: OBJECT';
var isEmpty, isString, toString, result:OBJECT';
var this'cons,this'cons',
this'left,this'left',
this'right,this'right': OBJECT';
var x,y,tsc,btc,bc,bus,ts,bt,busID,cl:OBJECT';
var instring,choicestring,dday,aday,dl,dt,d,at,p,adate,ddate:STRING;
var inint,xint,yint,zint,tSeats,aSeats,eSeats,busNo:INTEGER;
var object, choice, loginout:OBJECT;
Hierarchy#10010
Remote#6
M
Prime
CodePage#1
***#0
Call#20
Class#5
main
***
Minitours.main()#8
MinitoursInterface#74
Implements
Minitours#73
M
Prime
LoginGUI.new(tsc,bc,btc)
LoginGUI.login()
TravelSchemeContainer.new()
BusTourContainer.new()
BusContainer.new()

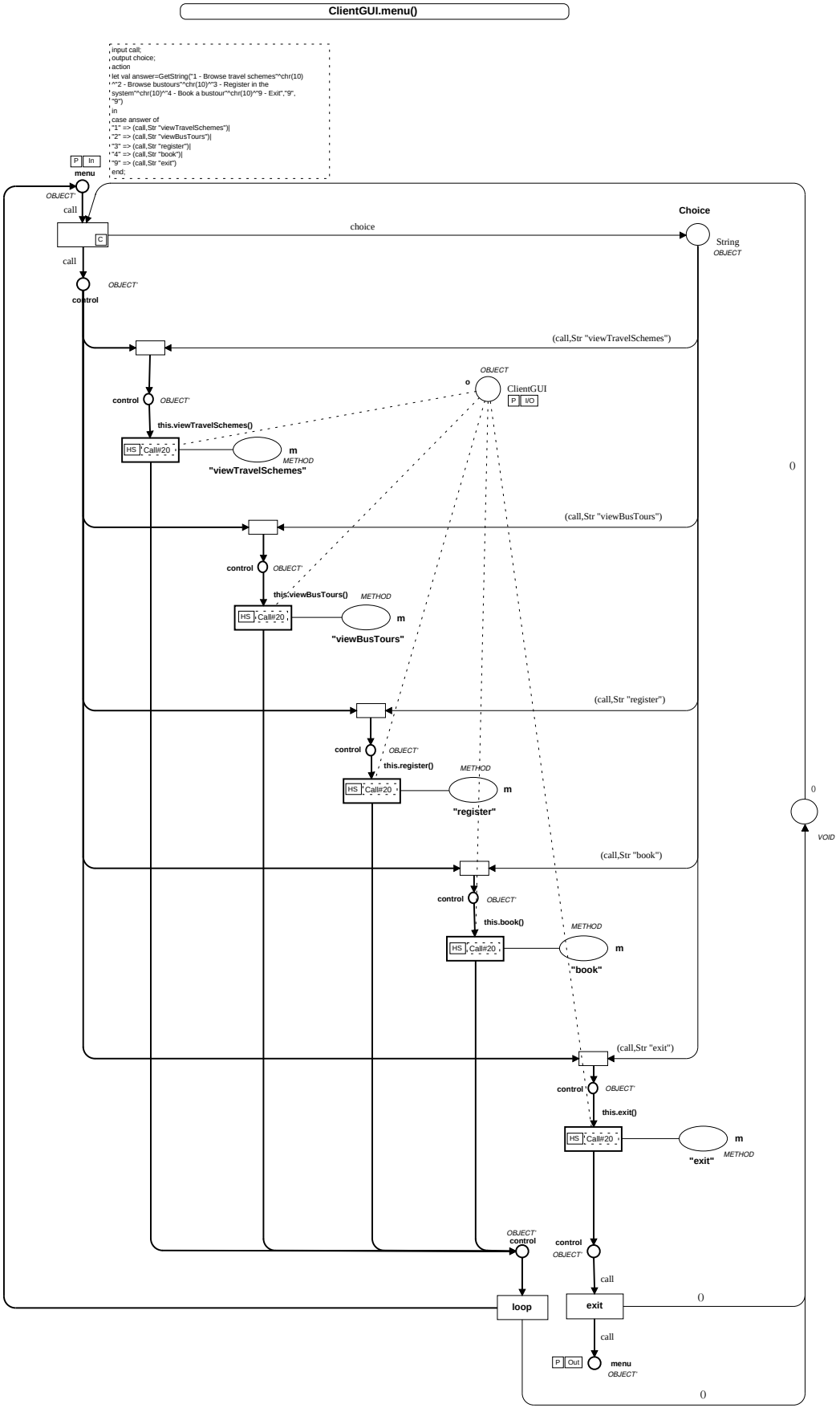
Remote-Page

```
action
 let val name = System.Unsafe.CInterface.gethostname () ^ ":" ^
     makestring (System.Unsafe.CInterface.getpid ());
  val (inf,outf) = my_execute("/home/tgi_3/tgi3/bin/messenger",[name]);
 in (my_pid := name; infile := inf; outfile := outf) end;
```



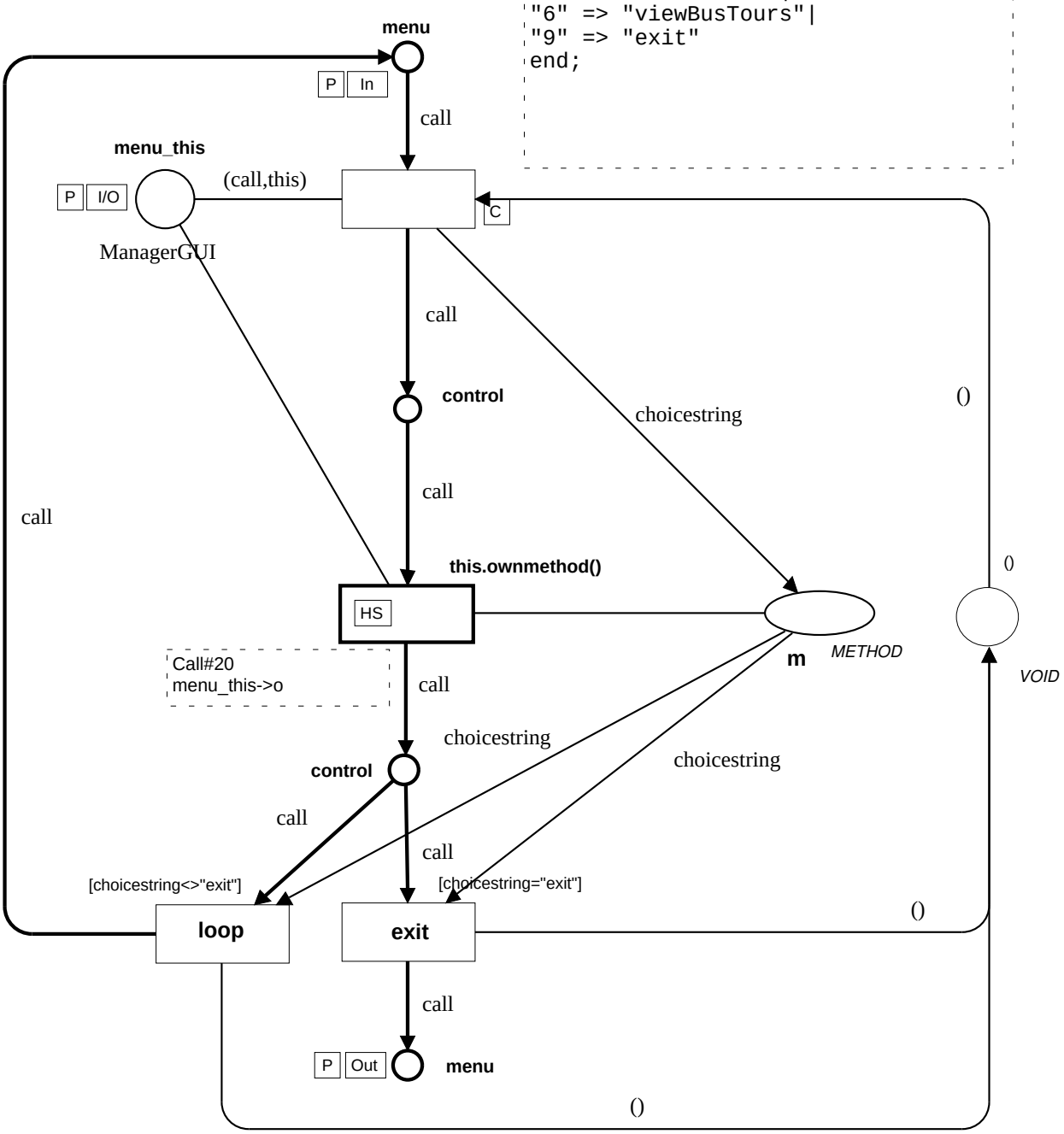

```
input (strmsg);
output (callmsgs,returnmsgs,stop);
action
 case toLocal (!my_pid) (strToMsg (strmsg)) of
    M(m)::rcv::param => (1‘(rcv::M(m)::param), empty, false)
  | I(i)::msg        => (empty,        1‘(I(i)::msg), false)
  | msg              => (empty,        1‘msg,         true);
```


Hierarchy#10010
Remote#6
M
Prime
CodePage#1
***#0
LoginGUIInterface#35
implements
LoginGUI#10
***
M
Prime
new
LoginGUI.new(TSC,BC,BTC)#3
login
LoginGUI.login()#54
ManagerGUI.new()
ClientGUI.new()
ManagerGUIInterface#36
implements
ManagerGUI#11
***
M
Prime
new
ManagerGUI.new()#13
this.menu()
menu
ManagerGUI.menu()#7
this.ownmethod()
defineTravel
ManagerGUI.defineTravelScheme()#27
TSC.createTravelScheme()
define
ManagerGUI.defineBusTour()#60
BTC.createBusTour()
defineBus
ManagerGUI.defineBus()#61
BC.createBus()
viewTravel
viewTravel
MGUI.viewTravelSchemes()#64
TSC.browse()
viewBusTours
viewBusTours
MGUI.viewBusTours()#67
BTC.browse()
viewBusses
MGUI.viewBusses()#69
BC.browse()
exit
ManagerGUI.exit()#56
Call#20
Class#5
ClientGUIInterface#37
implements
ClientGUI#12
***
M
Prime
this.viewTravelSchemes()
this.viewBusTours()
this.register()
this.book()
this.exit()
new
ClientGUI.new()#8
this.menu()
menu
ClientGUI.menu()#9
register
ClientGUI.register()#14
ClientProfile.new()
book
ClientGUI.book(ClientProfile,
BusTourContainer.makeReservation()
BusTourContainer.getBusTour(btno)
exit
ClientGUI.exit()#16

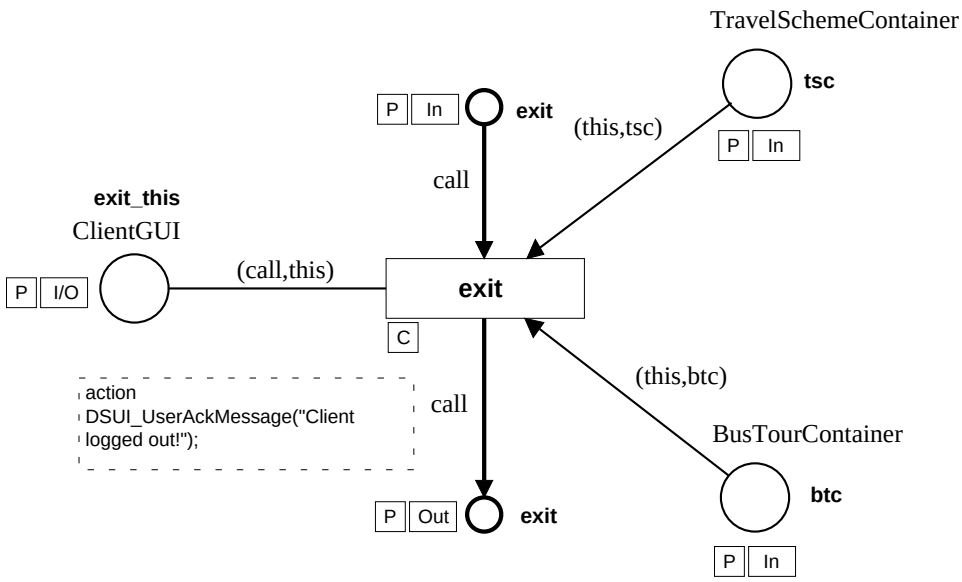

ClientGUI.menu()
input call;
output choice;
action
let val answer=GetString("1 - Browse travel schemes"^chr(10)
^"2 - Browse bustours"^chr(10)^"3 - Register in the
system"^chr(10)^"4 - Book a bustour"^chr(10)^"9 - Exit","9",
"9")
in
case answer of
"1" => (call,Str "viewTravelSchemes")|
"2" => (call,Str "viewBusTours")|
"3" => (call,Str "register")|
"4" => (call,Str "book")|
"9" => (call,Str "exit")
end;
P In
menu
OBJECT'
call
C
choice
Choice
String
OBJECT
call
OBJECT'
control
(call,Str "viewTravelSchemes")
OBJECT'
o
ClientGUI
P I/O
control OBJECT'
this.viewTravelSchemes()
HS Call#20
m
METHOD
"viewTravelSchemes"
()
(call,Str "viewBusTours")
control OBJECT'
this.viewBusTours()
METHOD
HS Call#20
m
"viewBusTours"
(call,Str "register")
control OBJECT'
this.register()
METHOD
HS Call#20
m
"register"
0
VOID
(call,Str "book")
control OBJECT'
this.book()
METHOD
HS Call#20
m
"book"
(call,Str "exit")
control OBJECT'
this.exit()
HS Call#20
m
METHOD
"exit"
OBJECT'
control
control
OBJECT'
call
loop
exit
()
call
P Out
menu
OBJECT'
()

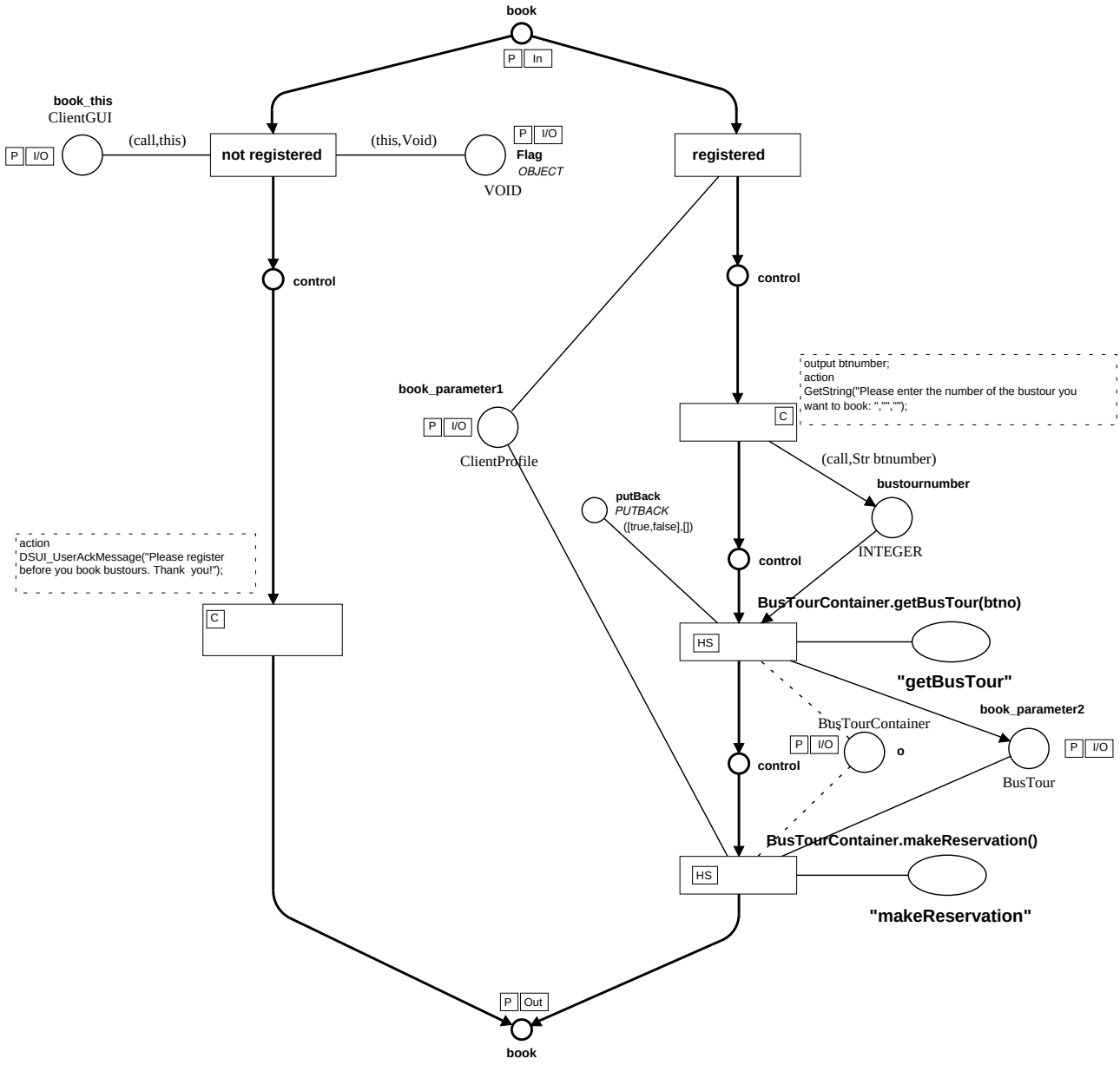
ManagerGUI.menu()

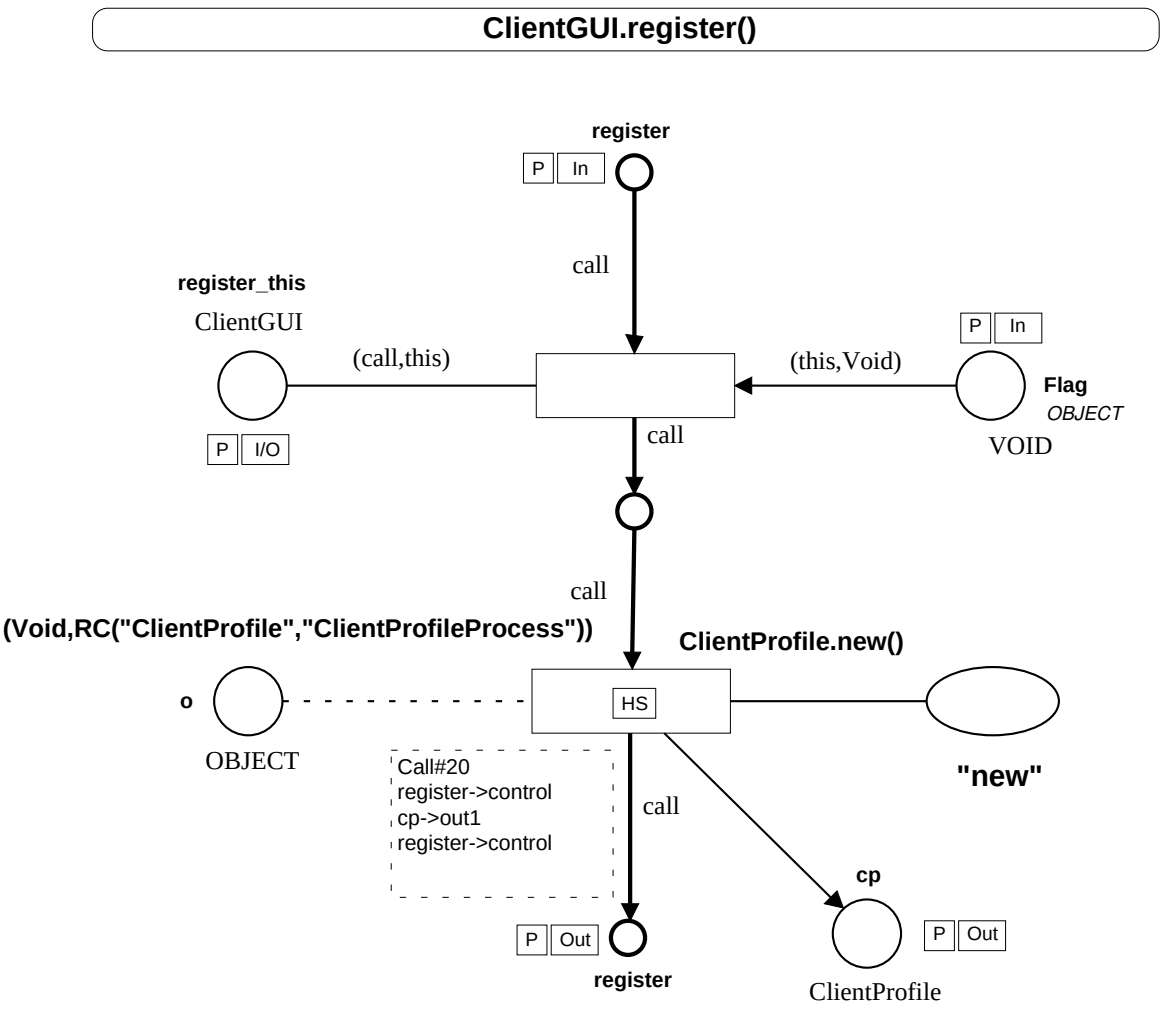
```
output choicestring;
action
let val answer=GetString("1 - Define
a travel scheme or 2 - Define a
bus"^chr(10)^"3 - Define a bustour
or 4 - Browse travel schemes"^chr(10)
^"5 - Browse busses or 6 - Browse
bustours"^chr(10)^"9 - Exit","9","9")
in
case answer of
"1" => "defineTravelScheme"|
"2" => "defineBus"|
"3" => "defineBusTour"|
"4" => "viewTravelSchemes"|
"5" => "viewBusses"|
"6" => "viewBusTours"|
"9" => "exit"
end;
```



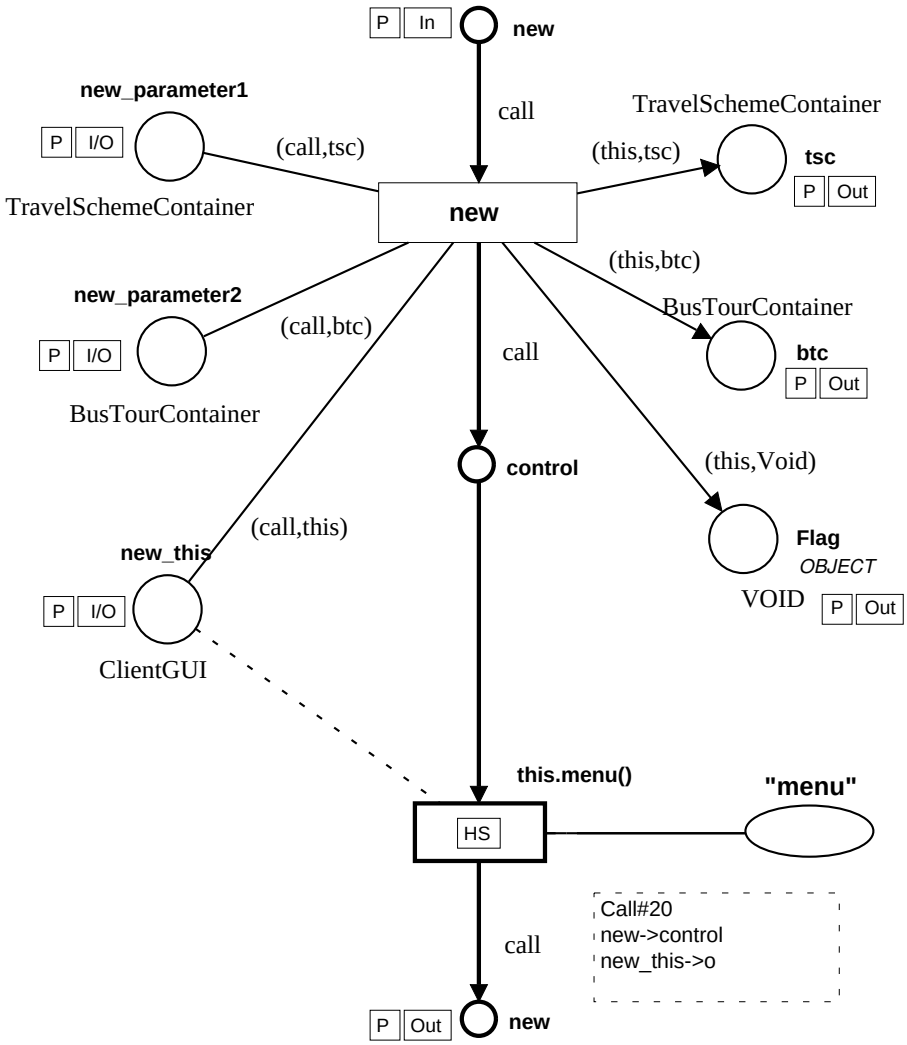
ClientGUI.exit()

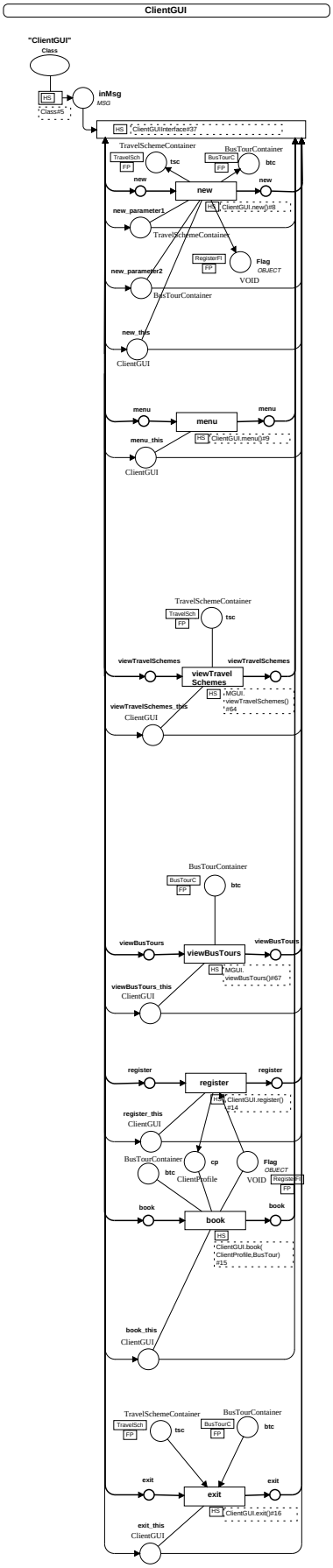


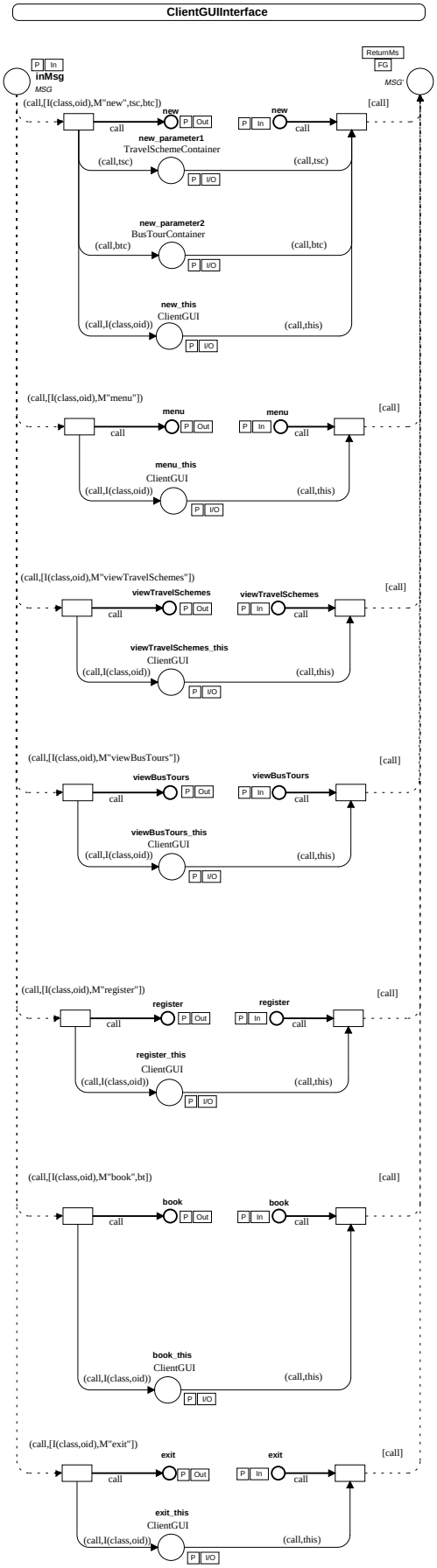

ClientGUI.book(cp:ClientProfile,bt:BusTour)
book
P In
book_this
ClientGUI
P I/O
(call,this)
not registered
(this,Void)
P I/O
Flag
OBJECT
VOID
registered
control
control
output btnumber;
action
GetString("Please enter the number of the bustour you want to book: ","","");
book_parameter1
P I/O
ClientProfile
C
(call,Str btnumber)
bustournumber
INTEGER
putBack
PUTBACK
([true,false],[])
control
BusTourContainer.getBusTour(btno)
action
DSUI_UserAckMessage("Please register before you book bustours. Thank you!");
C
HS
"getBusTour"
book_parameter2
BusTourContainer
P I/O
o
control
P I/O
BusTour
BusTourContainer.makeReservation()
HS
"makeReservation"
P Out
book


ClientGUI.register()
register
P
In
call
register_this
ClientGUI
(call,this)
(this,Void)
P
In
Flag
OBJECT
P
I/O
VOID
call
call
(Void,RC("ClientProfile","ClientProfileProcess"))
ClientProfile.new()
o
HS
OBJECT
"new"
Call#20
register->control
cp->out1
register->control
call
cp
P
Out
P
Out
register
ClientProfile

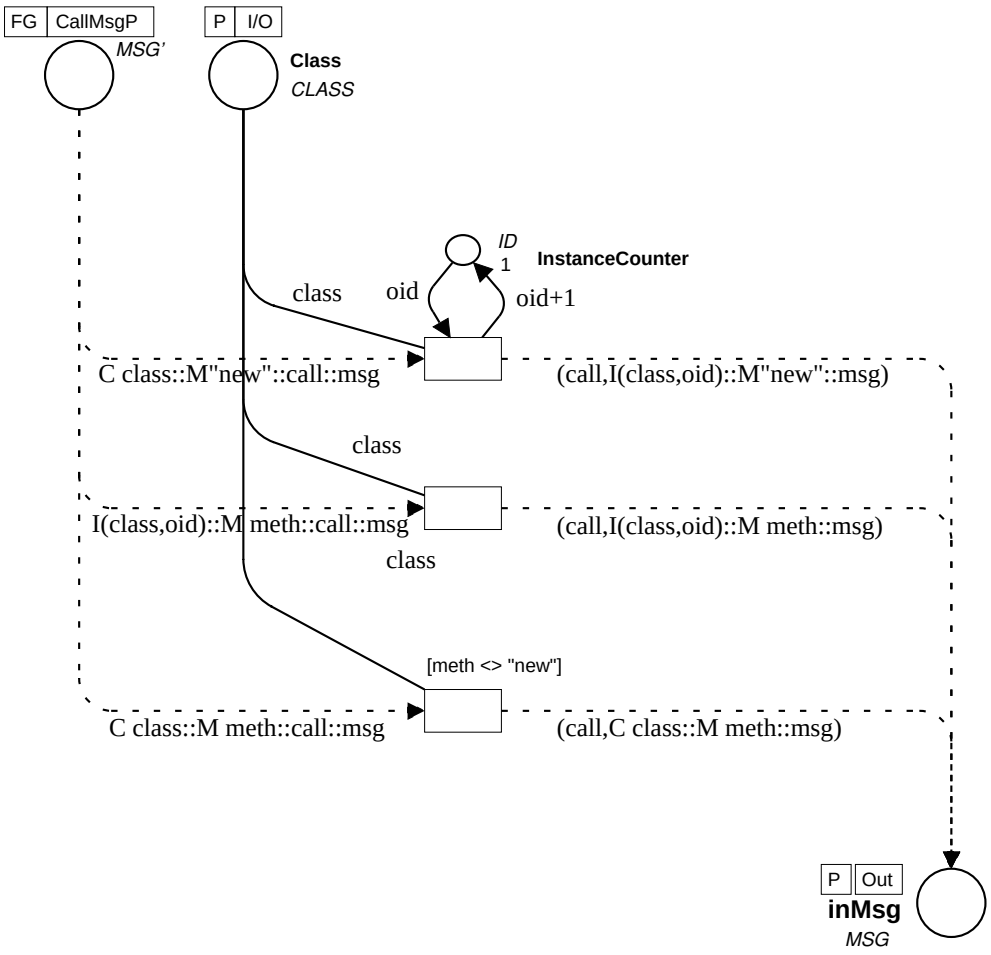
ClientGUI.new(tsc:TSC,btc:BTC)



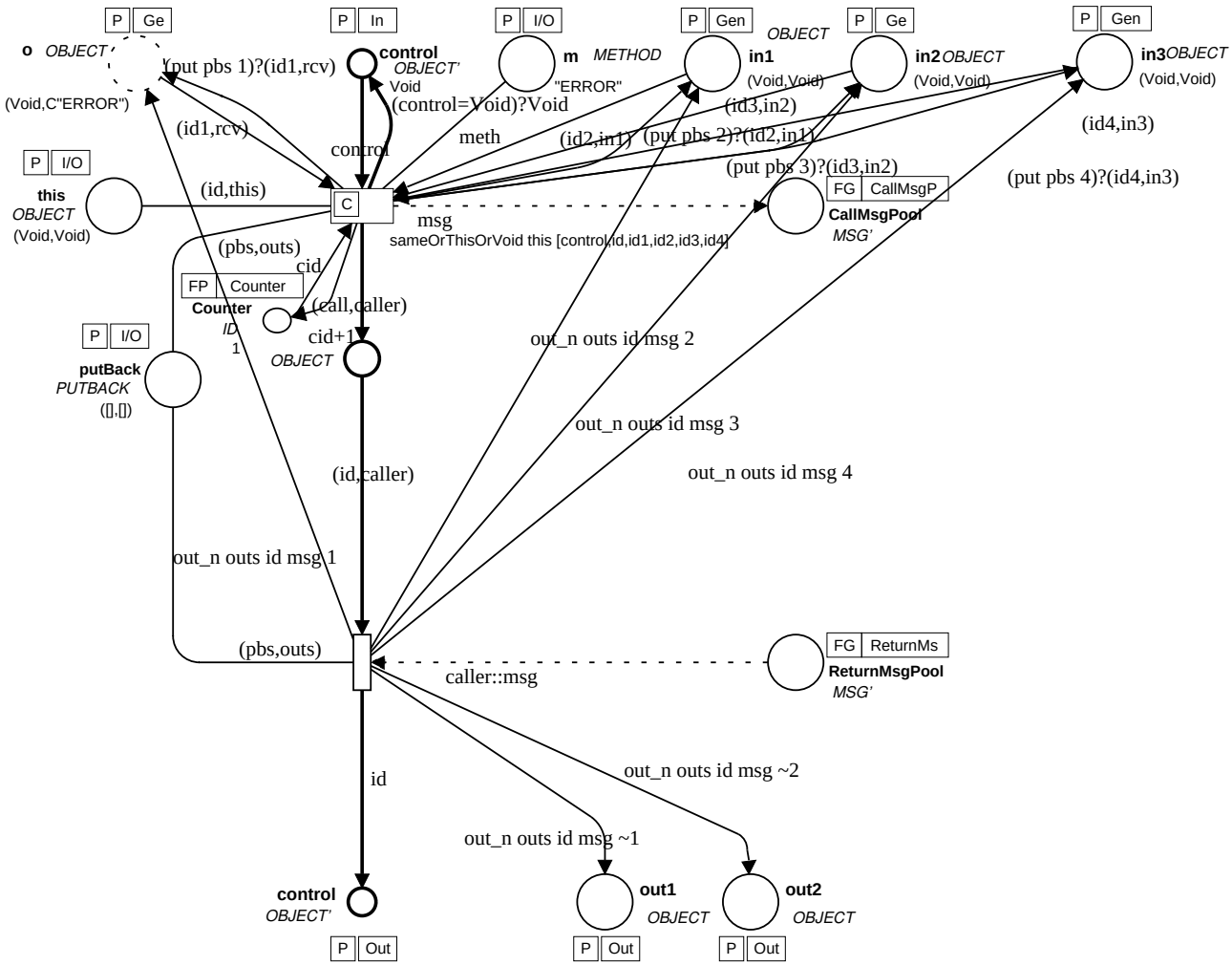

ClientGUI
"ClientGUI"
Class
inMsg
MSG
HS
Class#5
ClientGUIInterface37
TravelSchemeContainer
TravelSch
FP
tsc
BusTourContainer
BusTourC
btc
new
new_parameter1
new_parameter2
new_this
ClientGUI
HS
ClientGUI.new()#8
RegisterFl
Flag
OBJECT
VOID
menu
menu_this
ClientGUI.menu()#9
viewTravelSchemes
viewTravel Schemes
viewTravelSchemes_this
MGUI.viewTravelSchemes()#64
viewBusTours
viewBusTours
viewBusTours_this
MGUI.viewBusTours()#67
register
register
register_this
ClientGUI.register()#14
cp
ClientProfile
book
book
ClientGUI.book(ClientProfile,BusTour)#15
book_this
exit
exit
exit_this
ClientGUI.exit()#16


ClientGUIInterface
P In
inMsg
MSG
ReturnMs
FG
MSG'
(call,[I(class,oid),M"new",tsc,btc])
new
call
P Out
P In
new
call
[call]
new_parameter1
TravelSchemeContainer
(call,tsc)
(call,tsc)
P I/O
new_parameter2
BusTourContainer
(call,btc)
(call,btc)
P I/O
new_this
ClientGUI
(call,I(class,oid))
(call,this)
P I/O
(call,[I(class,oid),M"menu"])
menu
call
P Out
P In
menu
call
[call]
menu_this
ClientGUI
(call,I(class,oid))
(call,this)
P I/O
(call,[I(class,oid),M"viewTravelSchemes"])
viewTravelSchemes
call
P Out
P In
viewTravelSchemes
call
[call]
viewTravelSchemes_this
ClientGUI
(call,I(class,oid))
(call,this)
P I/O
(call,[I(class,oid),M"viewBusTours"])
viewBusTours
call
P Out
P In
viewBusTours
call
[call]
viewBusTours_this
ClientGUI
(call,I(class,oid))
(call,this)
P I/O
(call,[I(class,oid),M"register"])
register
call
P Out
P In
register
call
[call]
register_this
ClientGUI
(call,I(class,oid))
(call,this)
P I/O
(call,[I(class,oid),M"book",bt])
book
call
P Out
P In
book
call
[call]
book_this
ClientGUI
(call,I(class,oid))
(call,this)
P I/O
(call,[I(class,oid),M"exit"])
exit
call
P Out
P In
exit
call
[call]
exit_this
ClientGUI
(call,I(class,oid))
(call,this)
P I/O

Class-Page

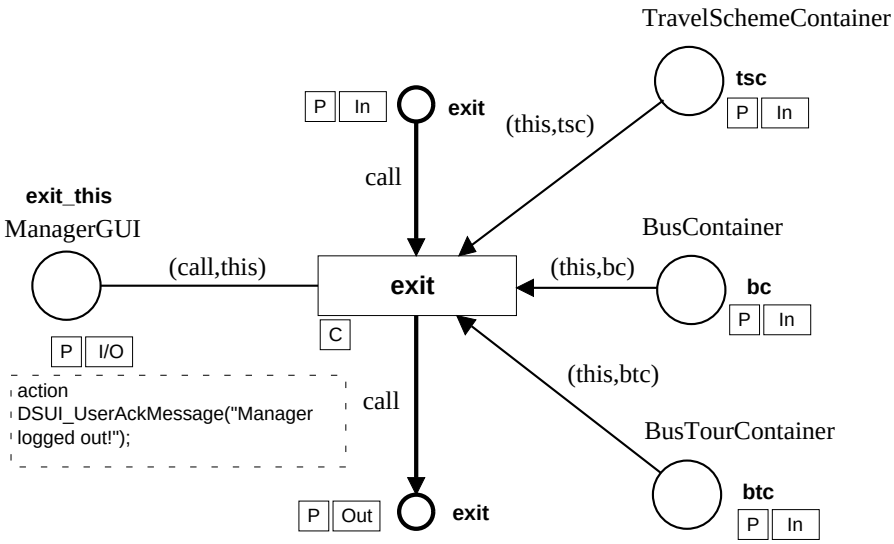

FG
CallMsgP
MSG'
P
I/O
Class
CLASS
ID
1
InstanceCounter
class
oid
oid+1
C class::M"new"::call::msg
(call,I(class,oid)::M"new"::msg)
class
I(class,oid)::M meth::call::msg
(call,I(class,oid)::M meth::msg)
class
[meth <> "new"]
C class::M meth::call::msg
(call,C class::M meth::msg)
P
Out
inMsg
MSG

Call-Page

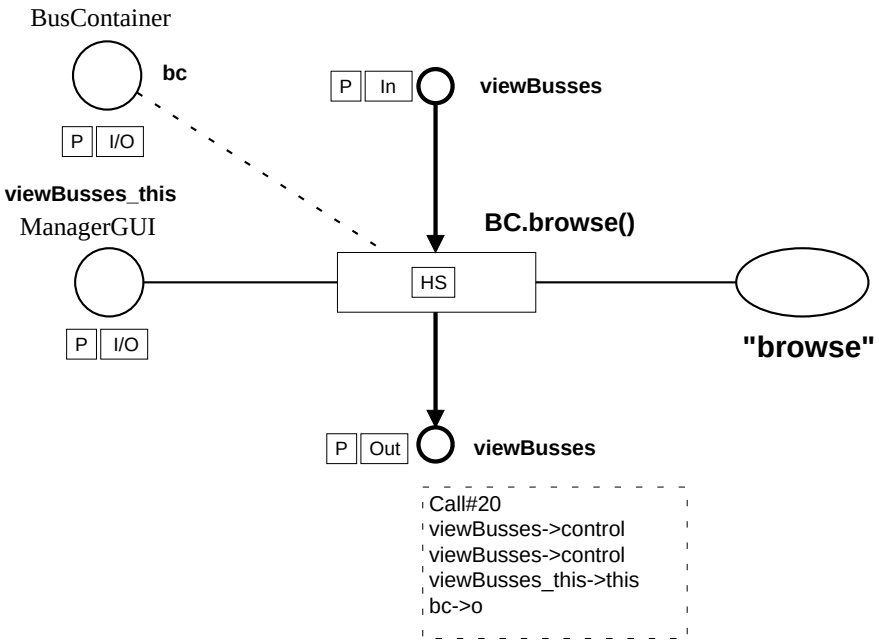



```
input (control,id,this,id1,rcv,id2,meth,id3,in1,in2,cid,in3,id4);
output(call,caller,msg);
action
  let val cntrl = nonThisOrVoid this [control,id,id1,id2,id3,id4];
    val caller = I("Call",cid);
in (cntrl,caller,rcv::M meth::caller::(noVoid [in1,in2,in3]))
end;
```

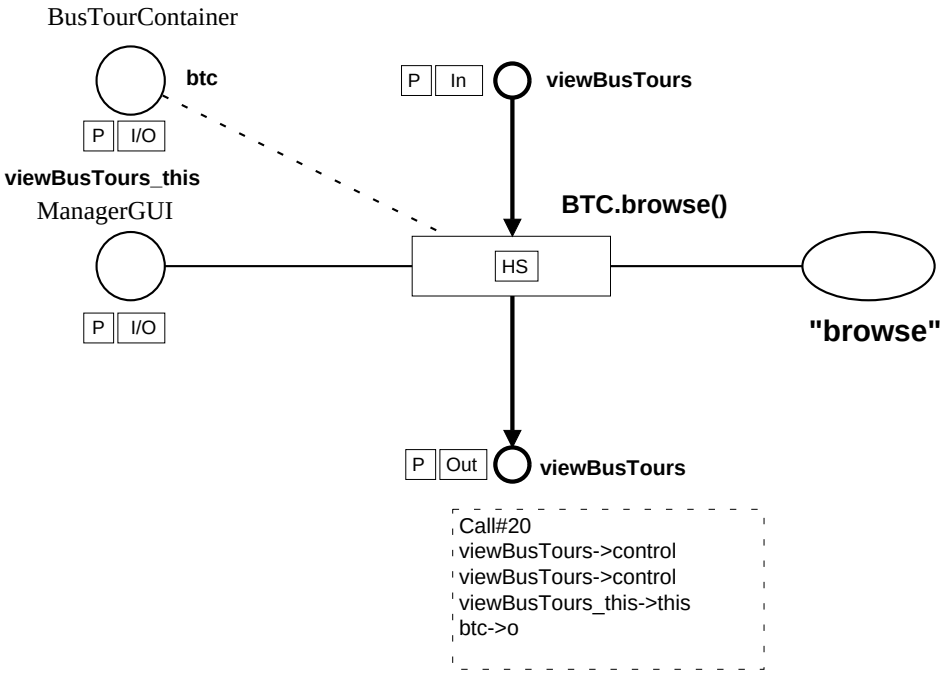
ManagerGUI.exit()



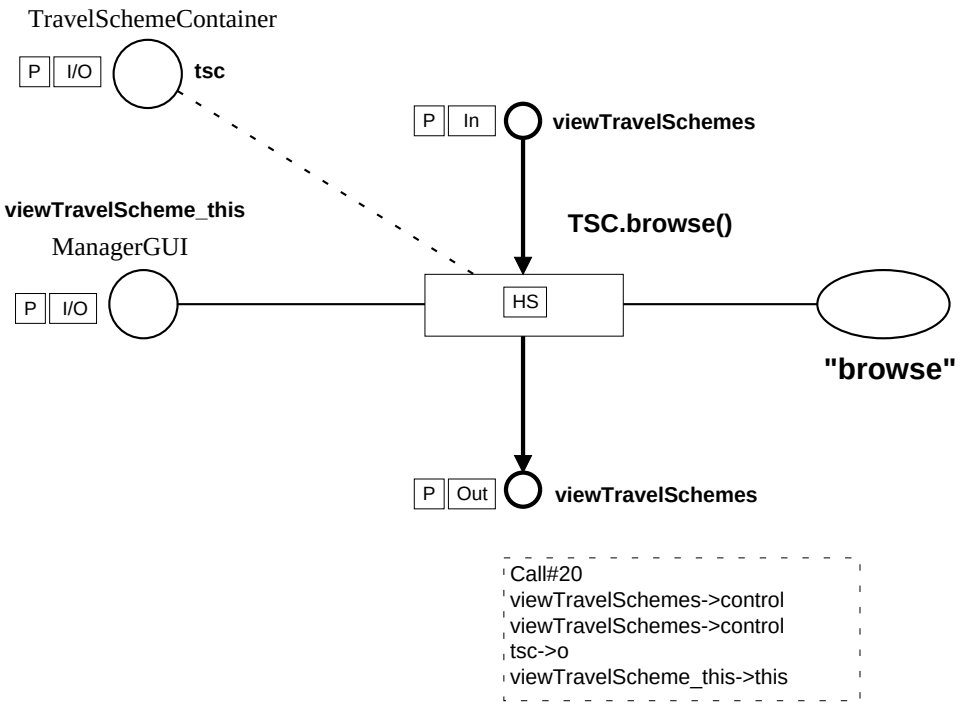
ManagerGUI.viewBusses()



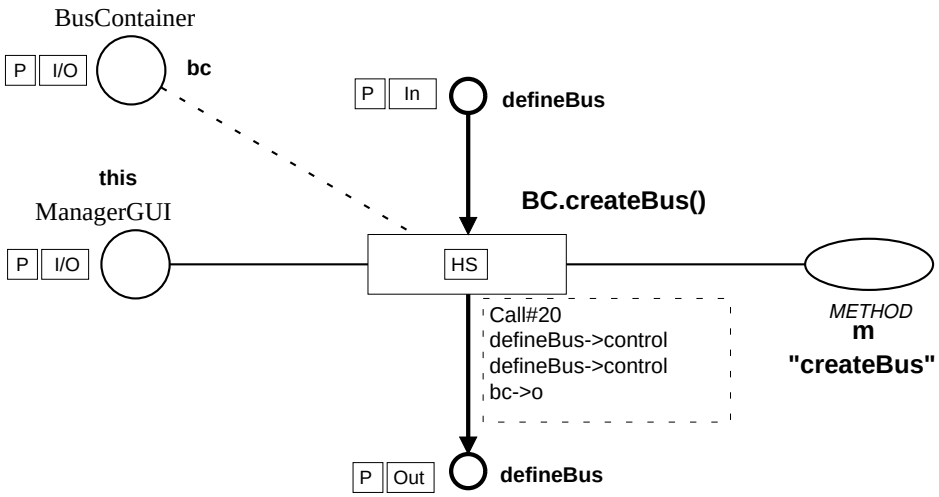
ManagerGUI.viewBusTours()



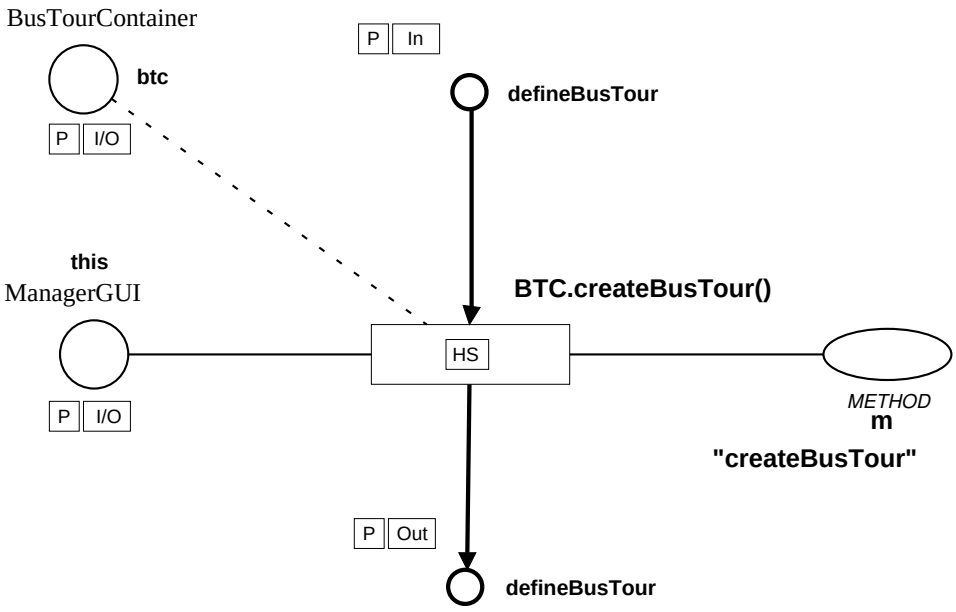
ManagerGUI.viewTravelSchemes()



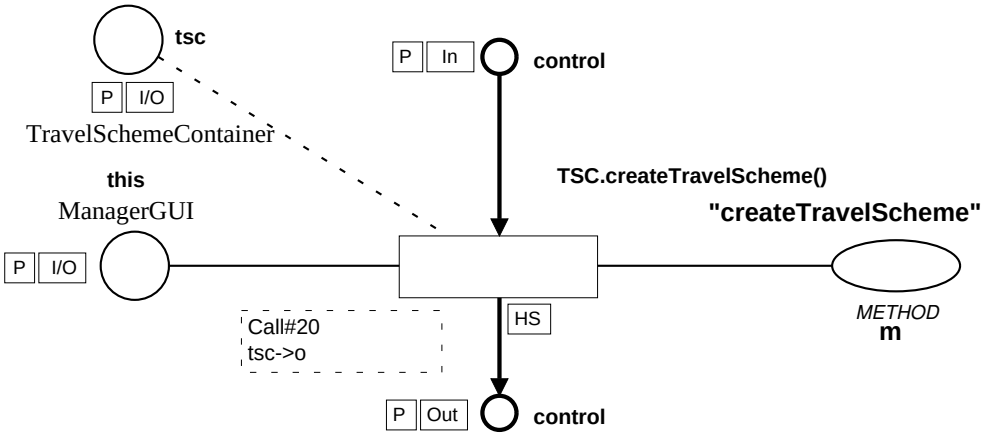
ManagerGUI.defineBus()



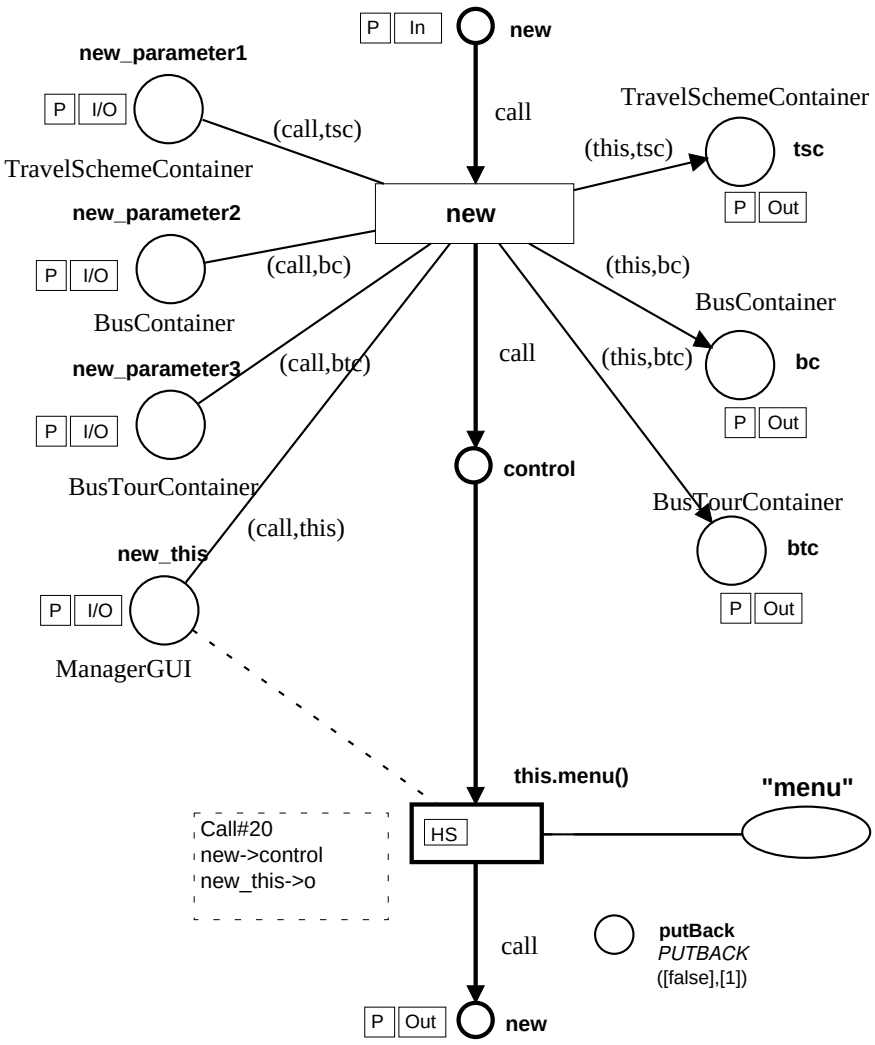
ManagerGUI.defineBusTour()

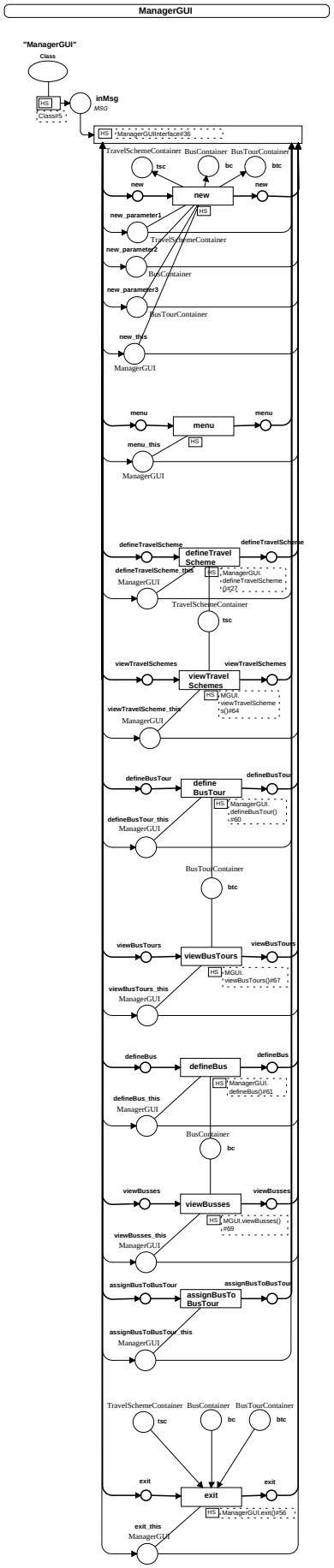


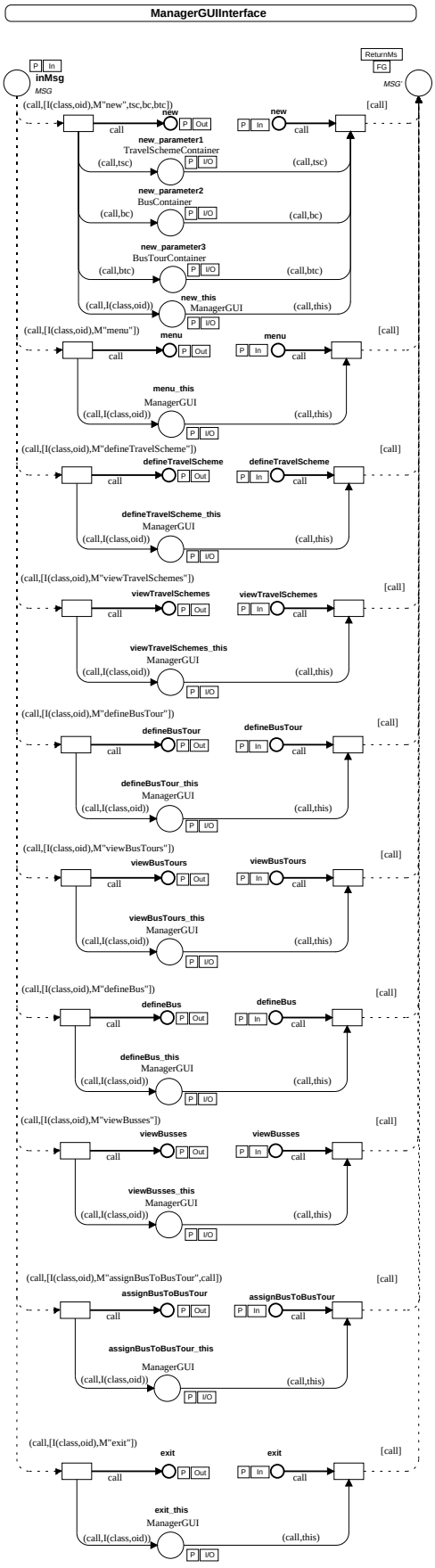
ManagerGUI.defineTravelScheme()

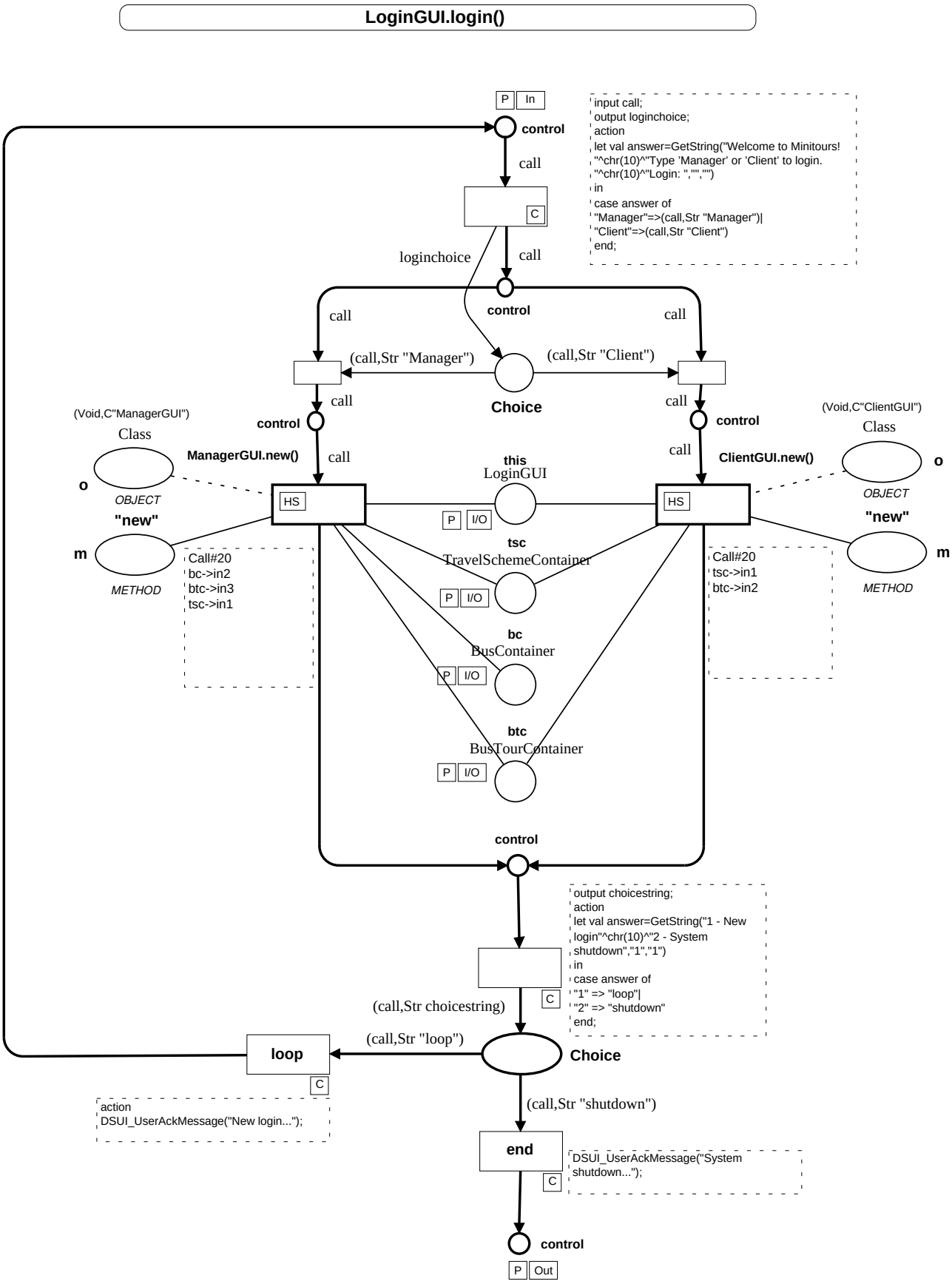


ManagerGUI.new(tsc:TSC,bc:BC,btc:BTC)

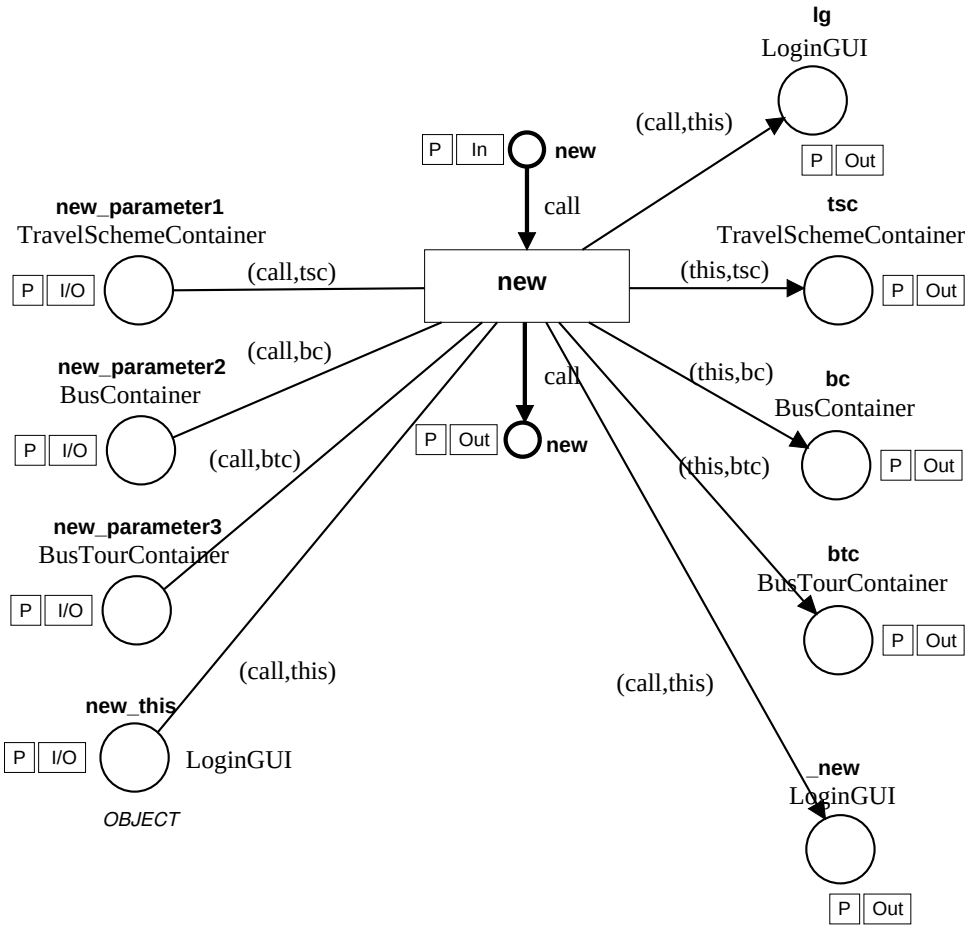


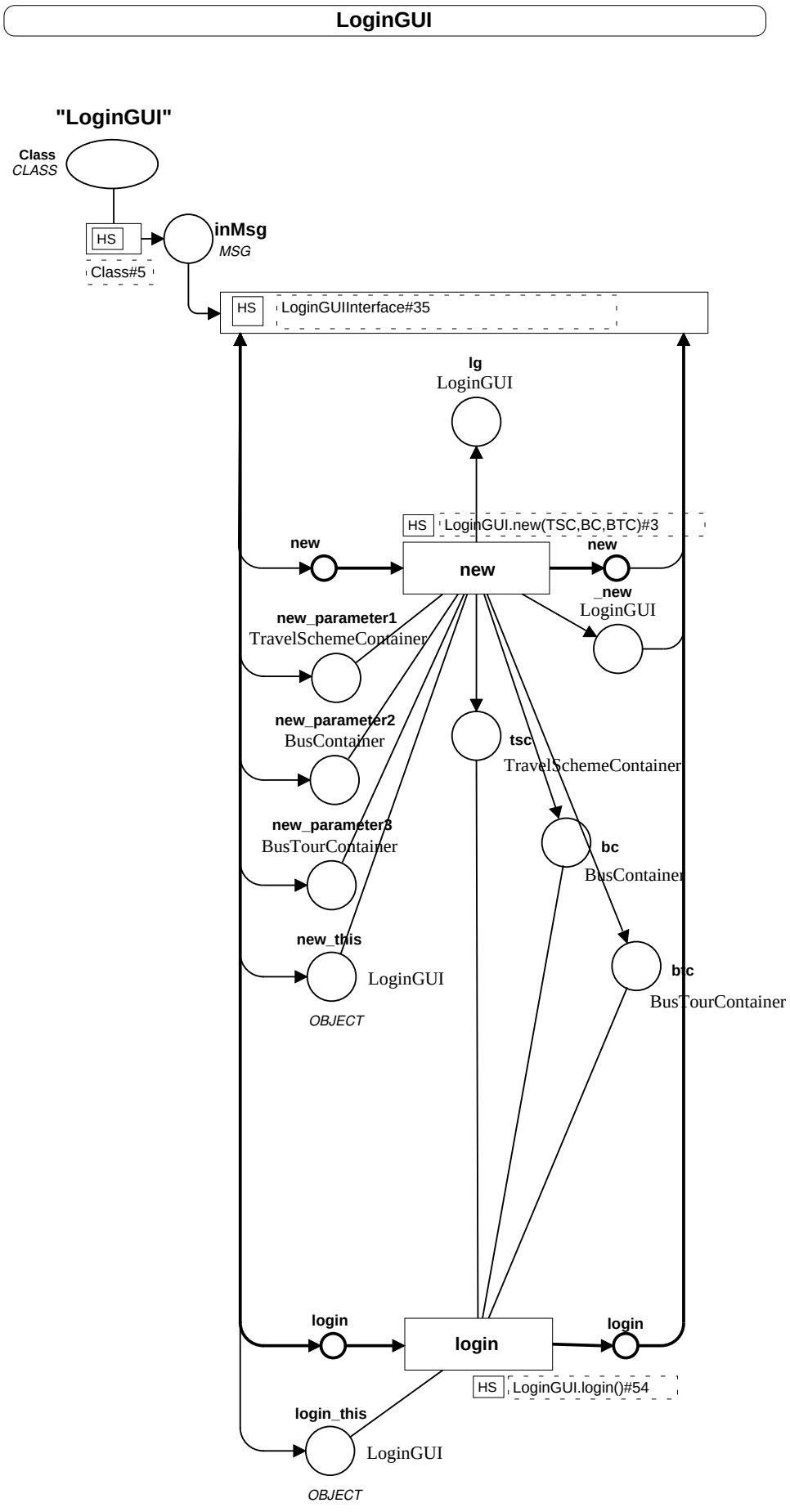

ManagerGUI
"ManagerGUI"
Class
inMsg
MSG
HS
Class#5
HS
ManagerGUIInterface#36
TravelSchemeContainer
BusContainer
BusTourContainer
tsc
bc
btc
new
new_parameter1
new_parameter2
new_parameter3
new_this
ManagerGUI
menu
menu_this
defineTravelScheme
defineTravelScheme_this
viewTravelSchemes
viewTravelScheme_this
defineBusTour
defineBusTour_this
viewBusTours
viewBusTours_this
defineBus
defineBus_this
viewBusses
viewBusses_this
assignBusToBusTour
assignBusToBusTour_this
exit
exit_this

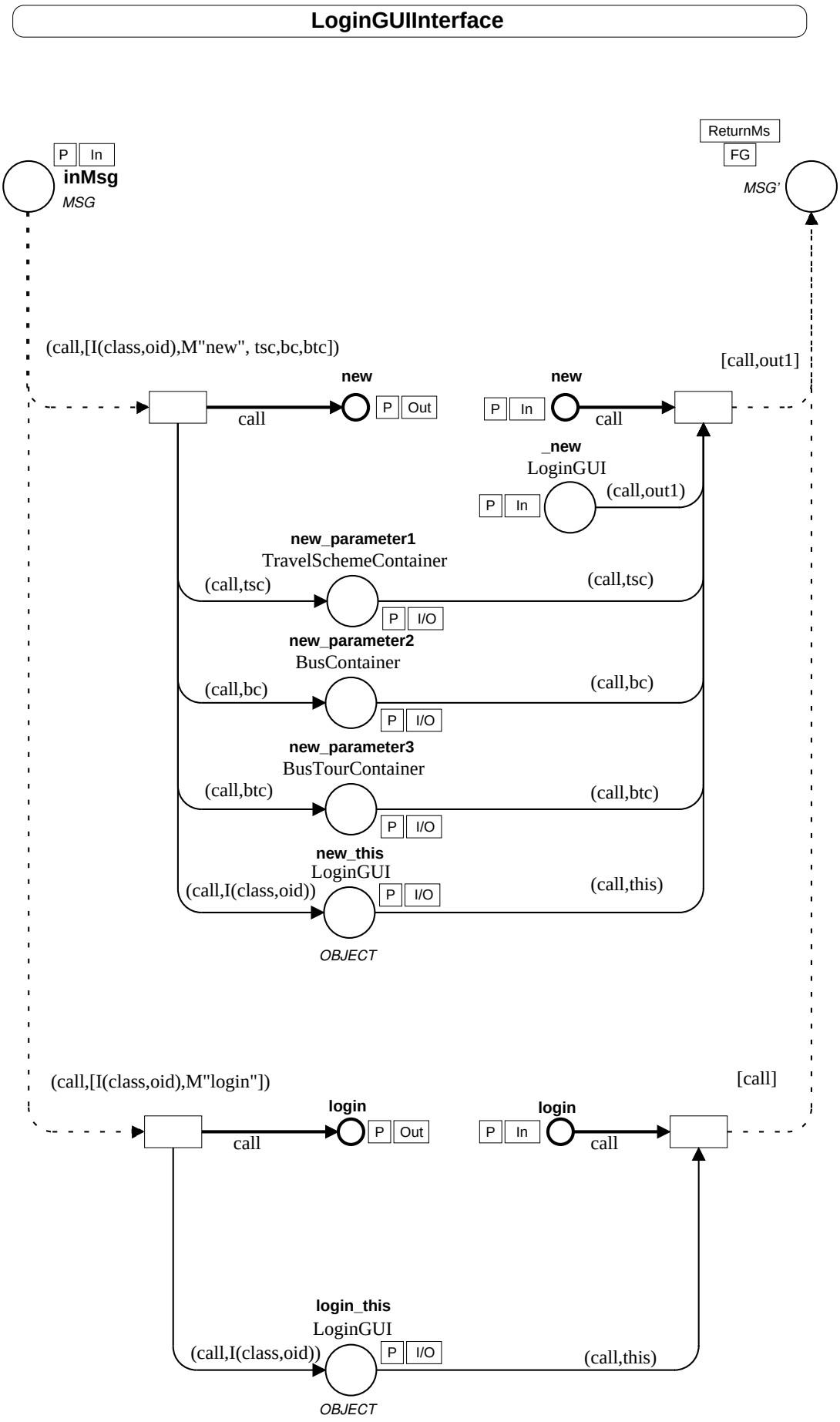

ManagerGUIInterface
P In
inMsg
MSG
ReturnMs
FG
MSG'
(call,[I(class,oid),M"new",tsc,bc,btc])
new
call
[call]
new_parameter1
TravelSchemeContainer
(call,tsc)
P I/O
new_parameter2
BusContainer
(call,bc)
new_parameter3
BusTourContainer
(call,btc)
new_this
ManagerGUI
(call,I(class,oid))
(call,this)
(call,[I(class,oid),M"menu"])
menu
menu_this
(call,[I(class,oid),M"defineTravelScheme"])
defineTravelScheme
defineTravelScheme_this
(call,[I(class,oid),M"viewTravelSchemes"])
viewTravelSchemes
viewTravelSchemes_this
(call,[I(class,oid),M"defineBusTour"])
defineBusTour
defineBusTour_this
(call,[I(class,oid),M"viewBusTours"])
viewBusTours
viewBusTours_this
(call,[I(class,oid),M"defineBus"])
defineBus
defineBus_this
(call,[I(class,oid),M"viewBusses"])
viewBusses
viewBusses_this
(call,[I(class,oid),M"assignBusToBusTour",call])
assignBusToBusTour
assignBusToBusTour_this
(call,[I(class,oid),M"exit"])
exit
exit_this


LoginGUI.login()
P
In
control
call
C
input call;
output loginchoice;
action
let val answer=GetString("Welcome to Minitours!
"^chr(10)^"Type 'Manager' or 'Client' to login.
"^chr(10)^"Login: ","","")
in
case answer of
"Manager"=>(call,Str "Manager")|
"Client"=>(call,Str "Client")
end;
loginchoice
call
control
call
call
(call,Str "Manager")
(call,Str "Client")
Choice
call
call
control
control
(Void,C"ManagerGUI")
Class
ManagerGUI.new()
call
this
LoginGUI
call
ClientGUI.new()
(Void,C"ClientGUI")
Class
o
OBJECT
"new"
m
METHOD
HS
HS
P
I/O
tsc
TravelSchemeContainer
bc
BusContainer
btc
BusTourContainer
Call#20
bc->in2
btc->in3
tsc->in1
Call#20
tsc->in1
btc->in2
control
output choicestring;
action
let val answer=GetString("1 - New
login"^chr(10)^"2 - System
shutdown","1","1")
in
case answer of
"1" => "loop"|
"2" => "shutdown"
end;
(call,Str choicestring)
(call,Str "loop")
loop
Choice
action
DSUI_UserAckMessage("New login...");
(call,Str "shutdown")
end
DSUI_UserAckMessage("System
shutdown...");
control
P
Out

LoginGUI.new(tsc:TSC,bc:BC,btc:BTC)




LoginGUI
"LoginGUI"
Class
CLASS
HS
Class#5
inMsg
MSG
HS
LoginGUIInterface#35
lg
LoginGUI
HS
LoginGUI.new(TSC,BC,BTC)#3
new
new
new
_new
LoginGUI
new_parameter1
TravelSchemeContainer
new_parameter2
BusContainer
tsc
TravelSchemeContainer
new_parameter3
BusTourContainer
bc
BusContainer
new_this
LoginGUI
OBJECT
btc
BusTourContainer
login
login
login
HS
LoginGUI.login()#54
login_this
LoginGUI
OBJECT


LoginGUIInterface
P
In
inMsg
MSG
ReturnMs
FG
MSG'
(call,[I(class,oid),M"new", tsc,bc,btc])
[call,out1]
new
call
P
Out
P
In
new
call
_new
LoginGUI
(call,out1)
P
In
new_parameter1
TravelSchemeContainer
(call,tsc)
(call,tsc)
P
I/O
new_parameter2
BusContainer
(call,bc)
(call,bc)
P
I/O
new_parameter3
BusTourContainer
(call,btc)
(call,btc)
P
I/O
new_this
LoginGUI
(call,I(class,oid))
(call,this)
P
I/O
OBJECT
(call,[I(class,oid),M"login"])
[call]
login
call
P
Out
P
In
login
call
login_this
LoginGUI
(call,I(class,oid))
(call,this)
P
I/O
OBJECT

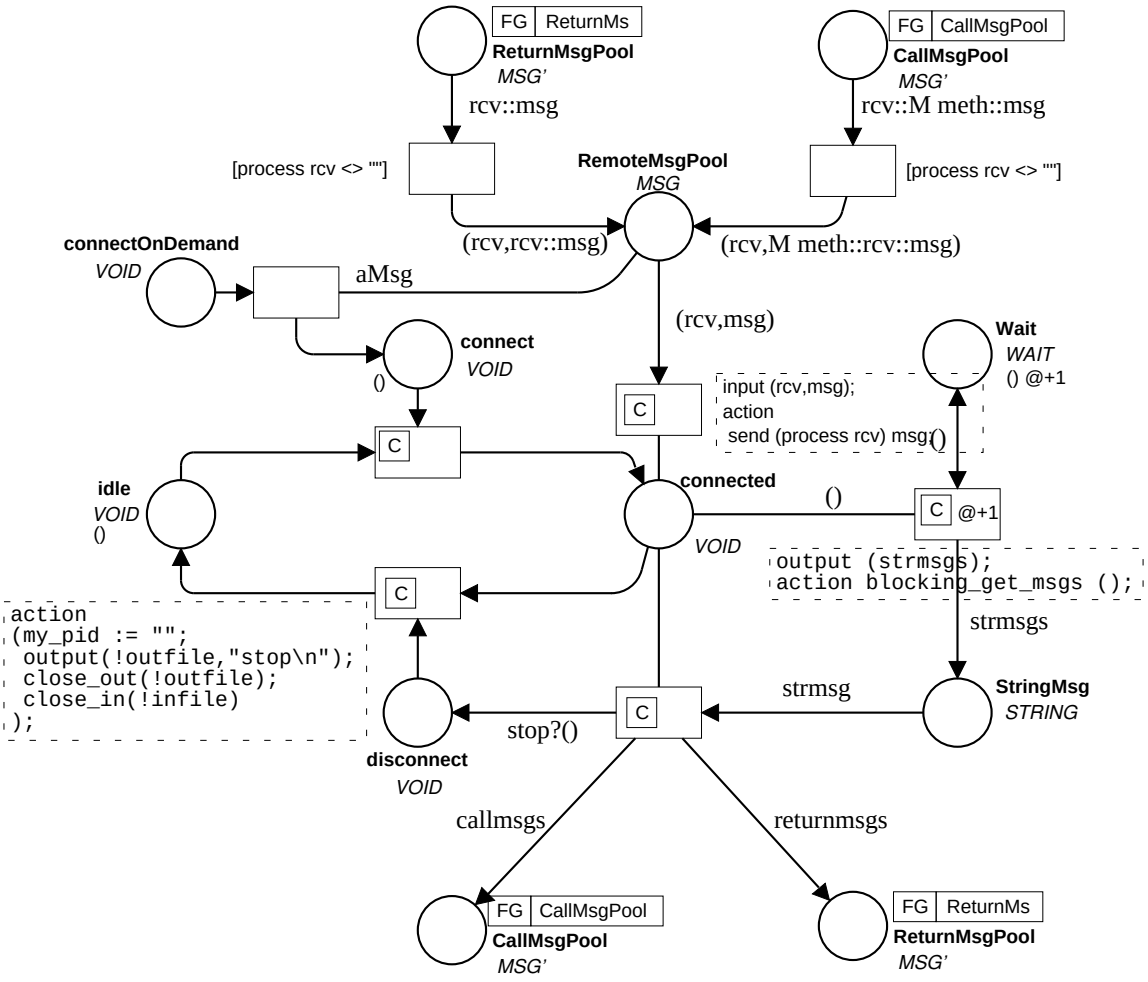
Code-Page

```
(* workaround: *)
val my_execute = execute;


color WAIT = unit timed;
color VOID = unit;
color INTEGER = int;
color REAL = real;
color STRING = string;
color BOOLEAN = bool;
color ID = INTEGER declare input_col;
color RID = STRING;
color BOOLEANLIST = list BOOLEAN;
color INTEGERLIST = list INTEGER;
color PUTBACK = product BOOLEANLIST * INTEGERLIST;
color VALUE = with Integer | RealNumber | String | Boolean;
color MODIFIER = with Static;
color MODIFIERS = list MODIFIER;
color CLASS = STRING;
color METHOD = STRING;
color INSTANCE = product CLASS * ID;
color RINSTANCE = product CLASS * RID * RID;
color RCLASS = product CLASS * RID;
color OBJECT' = union Void + Null
    + C:CLASS + RC:RCLASS
    + M:METHOD
    + I:INSTANCE + RI:RINSTANCE
    + Int:INTEGER + Str:STRING + Bool:BOOLEAN + Real:REAL
    declare mkst_col, of_M;
color MSG' = list OBJECT' declare input_col;
color TYPE = union Object + Reference + ClassRef + InstRef + Value
    + Method + AnyRef + Bottom
    + TC:CLASS + TI:CLASS + TM:METHOD + TV:VALUE;
color TYPELIST = list TYPE;
color MSGTYPE = product TYPELIST * TYPELIST;
color MSGDEF = product MODIFIERS * TYPELIST * METHOD * TYPELIST;
color RETURN' = product OBJECT' * TYPELIST;
color RETURN = product OBJECT'  * RETURN';
color MSG = product OBJECT'  * MSG';
color OBJECTLIST = MSG;
color OBJECT = product OBJECT'  * OBJECT';
color RIDxMSG' = product RID * MSG';

fun GetString(cr_prompt:string,cr_def:string,canc:string)=
let val Stringvalue =
DSUI_GetString{prompt=cr_prompt,def=cr_def}
in
  if Stringvalue <> "" then
       Stringvalue
  else
    GetString(cr_prompt,cr_def,canc)
  end
  handle _ => canc;

fun GetInteger(cr_prompt:string,cr_def:int,canc:int)=
let val Integervalue =
DSUI_GetIntegerValue{prompt=cr_prompt,def=cr_def}
in
  if Integervalue > 0 then
       Integervalue
  else
    GetInteger(cr_prompt,cr_def,canc)
  end
  handle _ => canc;



(* Net Inscription Abbreviations: *)
infix 5 ?';
fun p?v = if p then 1'v else empty;

infix 8 // ;
fun x // y = (x,y);

local fun mkst_list'      nil = ""
  | mkst_list' mkst [elem] = mkst elem
  | mkst_list' mkst (elem::lst) = mkst elem ^ "," ^ mkst_list' mkst lst
in fun mkst_list mkst lst = "[" ^ mkst_list' mkst lst ^ "]" end;

val msgToStr = mkst_list mkst_col'OBJECT';

fun strToMsg strmsg = (input_col'MSG' (open_string strmsg)) handle _ => [];

local fun remLocal' pid (C(class)) = RC(class,pid)
  | remLocal' pid (I(class,oid)) = RI(class,pid,makestring oid)
  | remLocal' _ old = old
in fun remLocal pid msg = map (remLocal' pid) msg end;

local fun toLocal' pid (old as RC(class,pid')) = if pid=pid' then C(class) else old
  | toLocal' pid (old as RI(class,pid',roid)) =
    if pid=pid' then I(class,input_col'ID (open_string roid)) else old
  | toLocal' _ old = old
in fun toLocal pid msg = map (toLocal' pid) msg end;


fun getType Void = Bottom
  | getType Null = AnyRef
  | getType (C class) = TC class
  | getType (RC(class,_)) = TC class
  | getType (M meth) = TM meth
  | getType (I(class,_)) = TI class
  | getType (RI(class,_,_)) = TI class
  | getType (Int _) = TV Integer
  | getType (Str _) = TV String
  | getType (Bool _) = TV Boolean
  | getType (Real _) = TV RealNumber;

infix 8 <== ;
fun type1  <== type2 =
 if type1 = type2 then true
 else case (type1,type2) of
   (_,      Object   ) => true
   | (Bottom, _      ) => true
   | (AnyRef, TC _   ) => true
   | (AnyRef, TI _   ) => true
   | (AnyRef, ClassRef ) => true
   | (AnyRef, InstRef  ) => true
   | (AnyRef, Reference) => true
   | (TC _  , ClassRef ) => true
   | (TC _  , Reference) => true
   | (TI _  , InstRef  ) => true
   | (TI _  , Reference) => true
   | (TM _  , Method   ) => true
   | (TV _  , Value    ) => true
   | (_  _  , _        ) => false;

fun ofType typ obj = getType obj <== typ;
val ofUser = ofType (TI "User");
val oftail = ofType (TI "tail");
val ofPair = ofType (TI "Pair");


fun typeCheck nil nil = true
  | typeCheck (typ::types) (obj::objs) =
 ofType typ obj andalso typeCheck types objs
  | typeCheck _ _ = false;

fun nonThisOrVoid    nil = Void (* darf nicht vorkommen! *)
  | nonThisOrVoid this (x::objs) = if x=Void orelse x=this then nonThisOrVoid this objs
          else x;

local
fun sameOrThisOrVoid' y _ nil = y <> Void
  | sameOrThisOrVoid' Void this (x::objs) =
  if x=Void orelse x=this then sameOrThisOrVoid' Void this objs
          else sameOrThisOrVoid' x this objs
  | sameOrThisOrVoid' y this (x::objs) =
  if x=Void orelse x=this orelse x=y then sameOrThisOrVoid' y this objs
          else false
in val sameOrThisOrVoid = sameOrThisOrVoid' Void end;

fun noVoid nil = nil
  | noVoid (Void::xs) = noVoid xs
  | noVoid (x::xs) = x::noVoid xs;

fun prefix xs 0 = nil
  | prefix (x::xs) n = x::prefix xs (n-1)
  | prefix nil _ = nil;

fun put (x::xs) 1 = x
  | put (x::xs) n = put xs (n-1)
  | put nil _ = true;

fun in_n call (obj::objs) 1 = 1'(call,obj)
  | in_n call (_::objs) n = in_n call objs (n-1)
  | in_n call nil _ = empty; (* 1'(call,Null); *)

local fun position' _  nil _ = 0
  | position' n (x::xs) y = if x = y then n else position' (n+1) xs y
in val position = position' 1
end;

fun out_n outs call msg index =
 if mem outs index
  orelse (length outs < -index andalso length msg >= -index)
 then let val pos = if mem outs index then position outs index
          else -index
   in 1'(call,nth(msg,pos-1)) end
 else empty;

fun process (RC(_,pid)) = pid
  | process (RI(_,pid,_)) = pid
  | process _ = "";

globref my_pid = "";
globref infile = std_in;
globref outfile = std_out;

fun send pid msg =
 (output(!outfile, "send\n"^ pid ^"\n" ^
  msgToStr(remLocal (!my_pid) msg) ^"\n");
    flush_out(!outfile));

val callcnt = ref 0;

infix 8 \;
fun (obj\meth) params =
  let val caller = I("CALL",(callcnt:= (!callcnt)+1;!callcnt))
  in (send (process (hd (remLocal (!my_pid) [obj])))
           (M meth::obj::caller::params);
           caller)
  end;

infix 8 return;
fun (obj return params) =
  (send (process (hd (remLocal (!my_pid) [obj])))
        (obj::params));

fun blocking_get_msgs () =
 (1_input_line(!infile)) !!
    (if can_input(!infile)>0
  then blocking_get_msgs ()
  else empty);
```

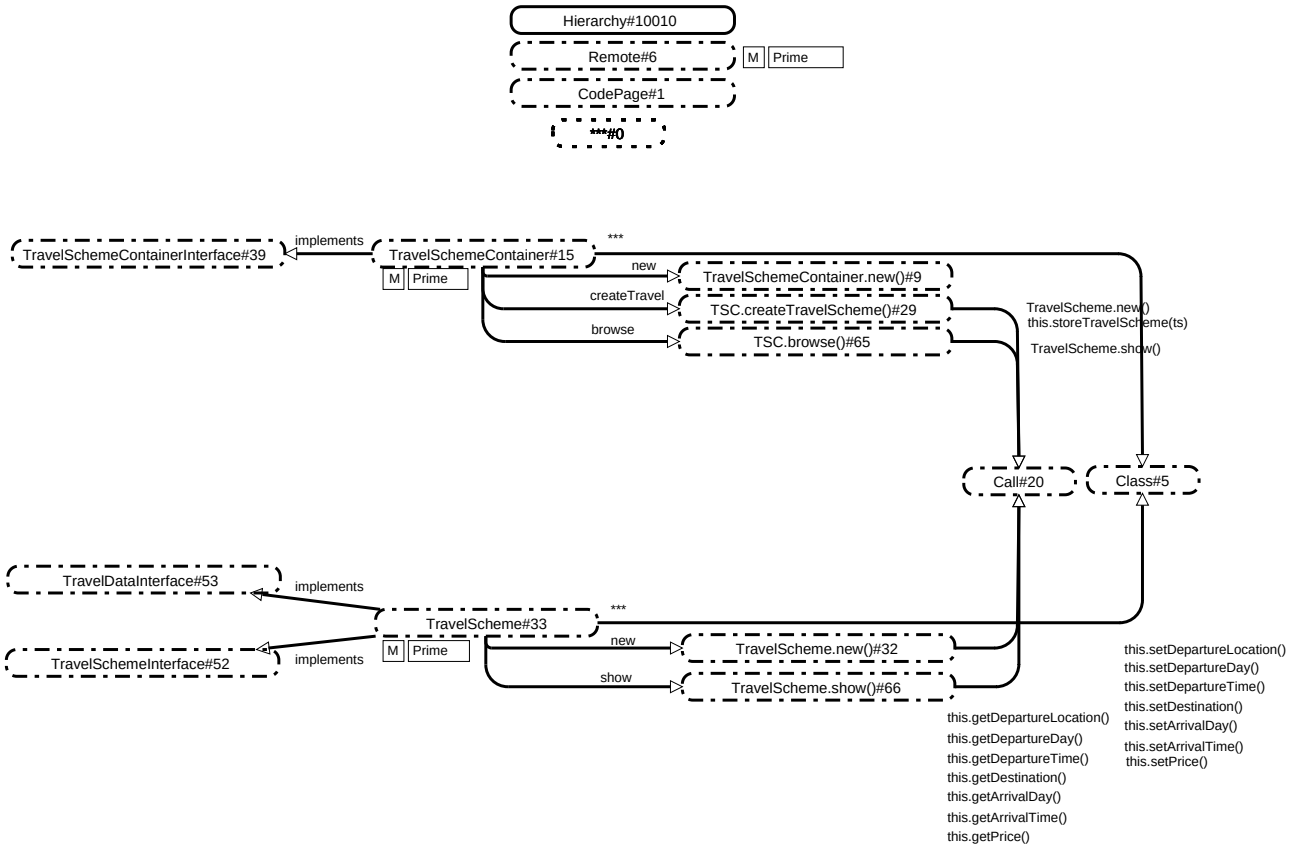
```
var localmsgs, callmsgs, returnmsgs: MSG' ms;
var strmsg: STRING;
var strmsgs: STRING ms;
var timer, timer': INTEGER;
var pid,roid: RID;
var pbs: BOOLEANLIST;
var outs: INTEGERLIST;
var msg: MSG';
var aMsg: MSG;
var onDemand, stop: BOOLEAN;
var class: CLASS;
var msgType: MSGTYPE;
var inClasses, outClasses: TYPELIST;
var cid,oid: ID;
var control,call,caller,
    id, id' ,id1, id2, id3, id4: OBJECT';
var meth: METHOD;
var obj, in1,in2,in3,out1,out2: OBJECT';
var this,classobj,snd,rcv,
 lst,pair,cons,tail,add,
     filter,filteredList,
 mapObj,filterObj: OBJECT';
var newHead: OBJECT';
var head,car,cdr, parameter, parameter1, parameter2: OBJECT';
var isEmpty, isString, toString, result:OBJECT';
var this'cons,this'cons',
 this'left,this'left',
 this'right,this'right': OBJECT';

var x,y,tsc,btc,bc,bus,ts,bt,busID,cl:OBJECT';
var instring,choicestring,dday,aday,dl,dt,d,at,p,adate,ddate,btnumber:
STRING;
var inint,xint,yint,zint,tSeats,aSeats,eSeats,busNo,dummy:INTEGER;
var object, choice, loginchoice:OBJECT;
```

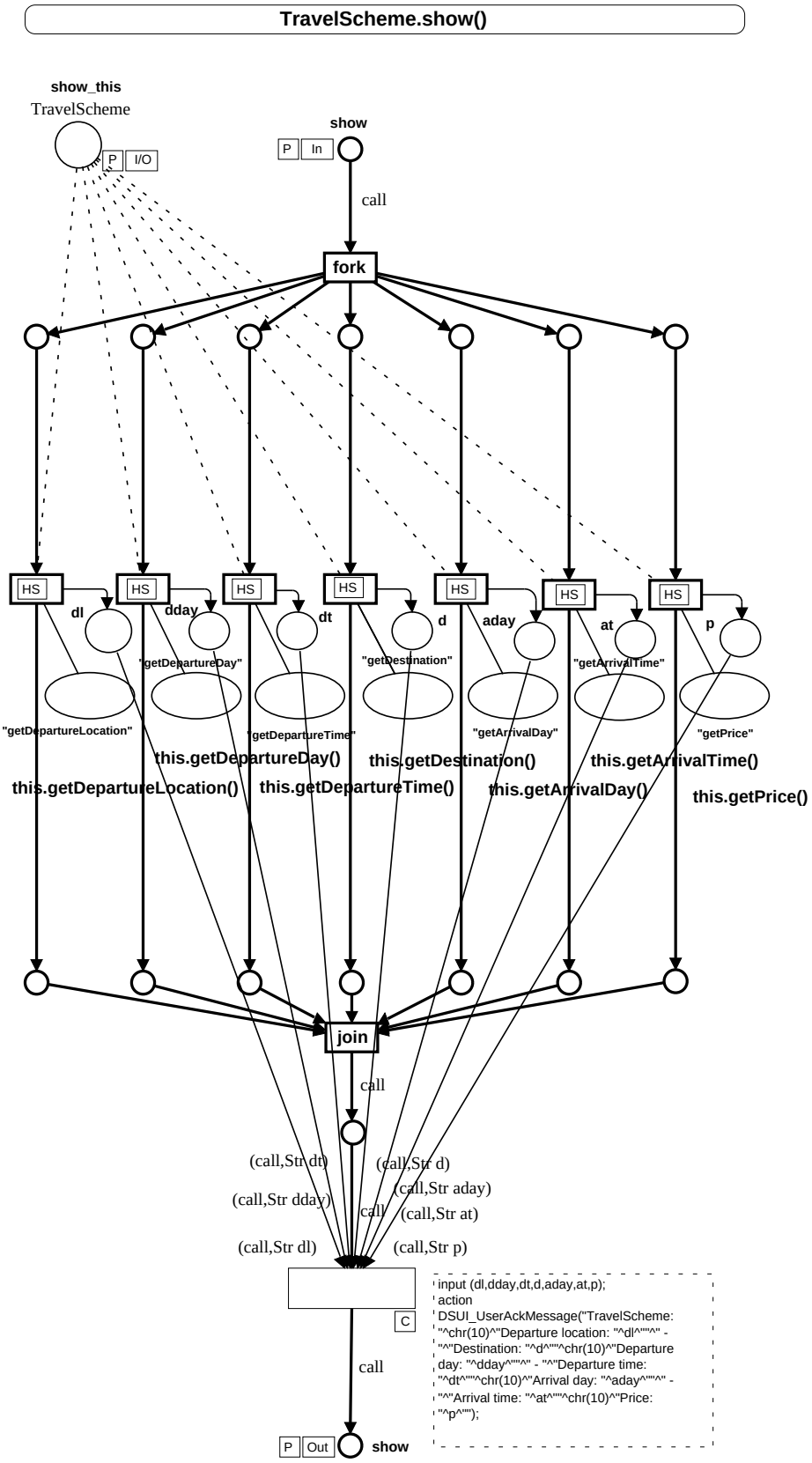
Remote-Page

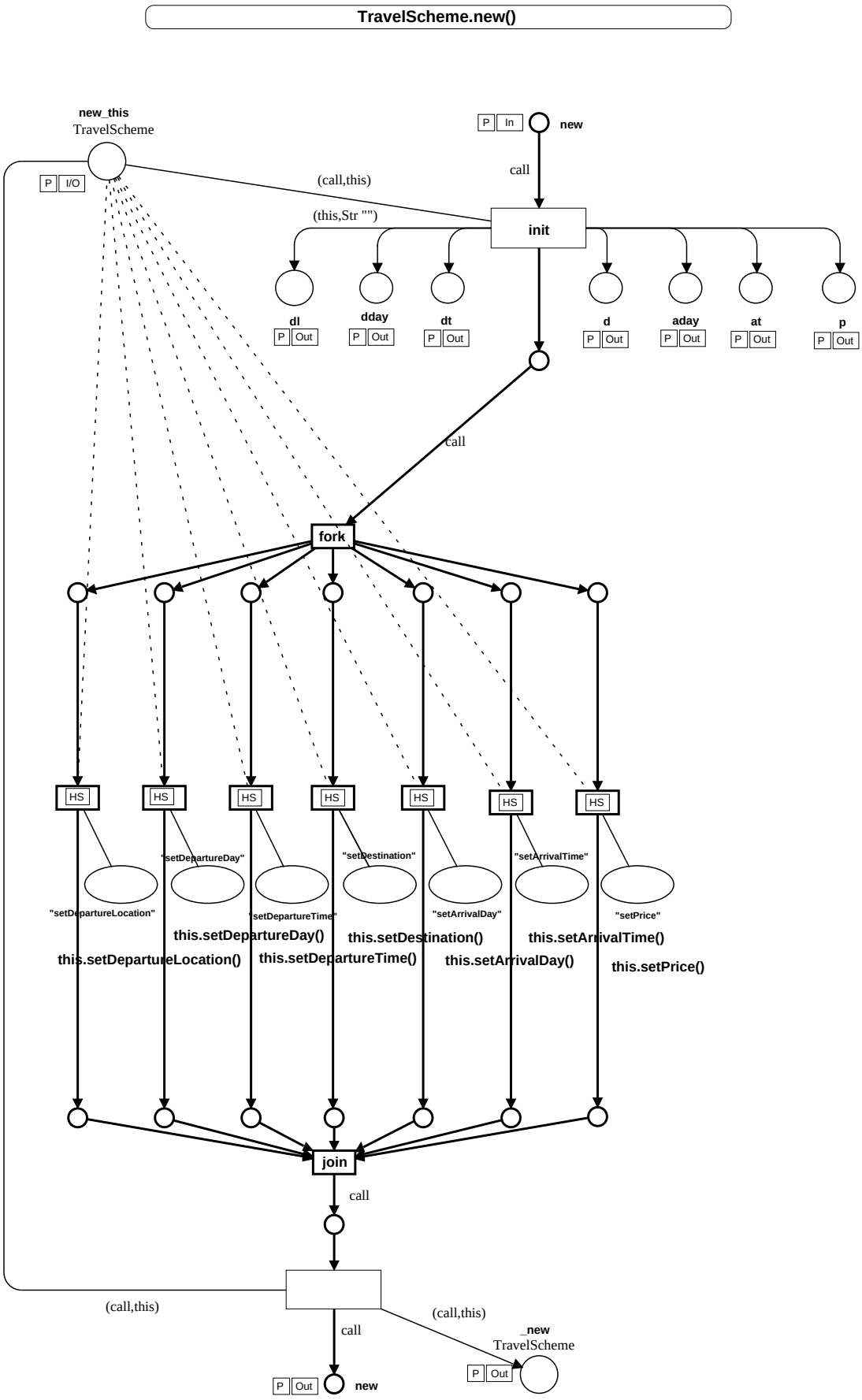
```
action
 let val name = "GUIProcess";
  val (inf,outf) = my_execute("/home/tgi_3/tgi3/bin/messenger",[name]);
 in (my_pid := name; infile := inf; outfile := outf) end;
```

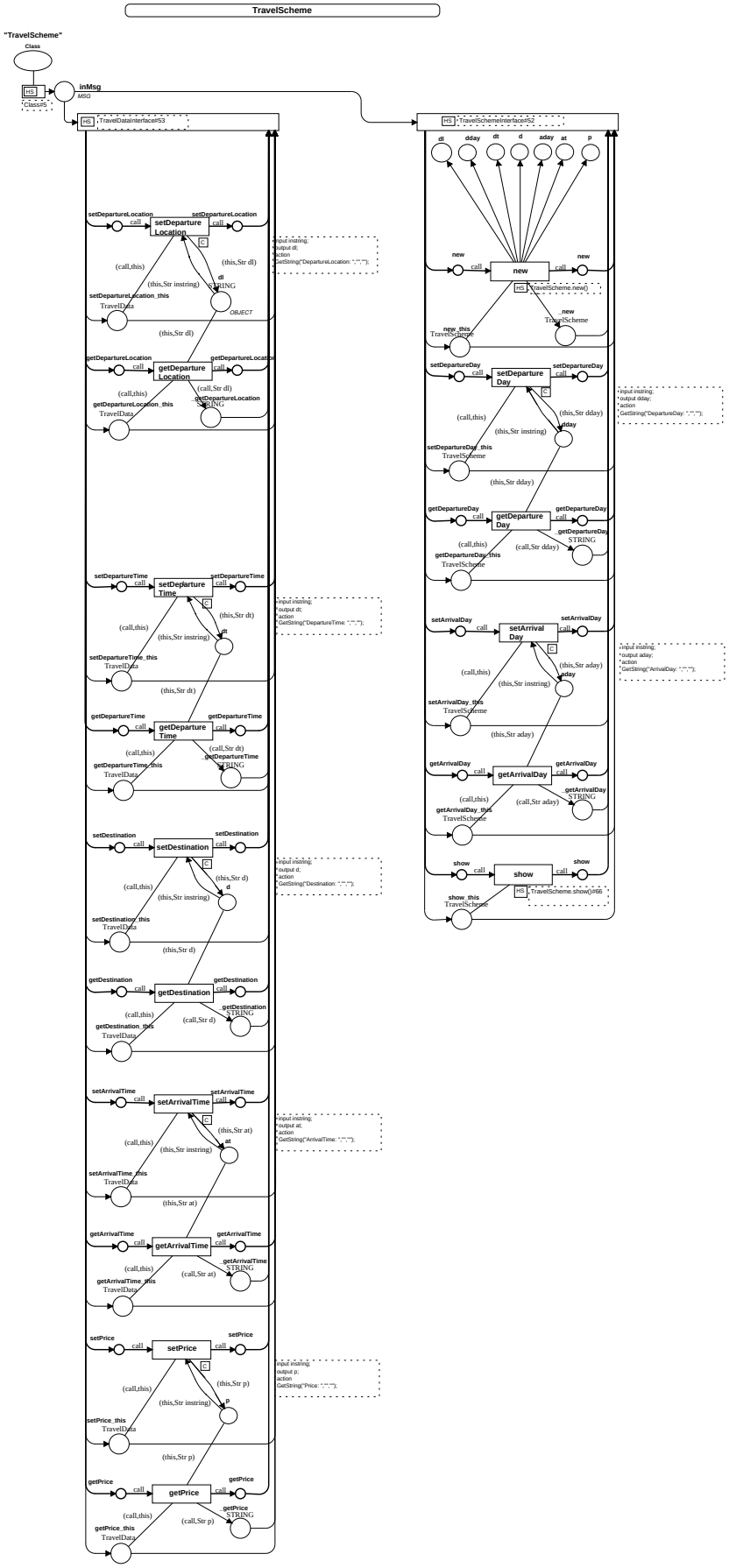


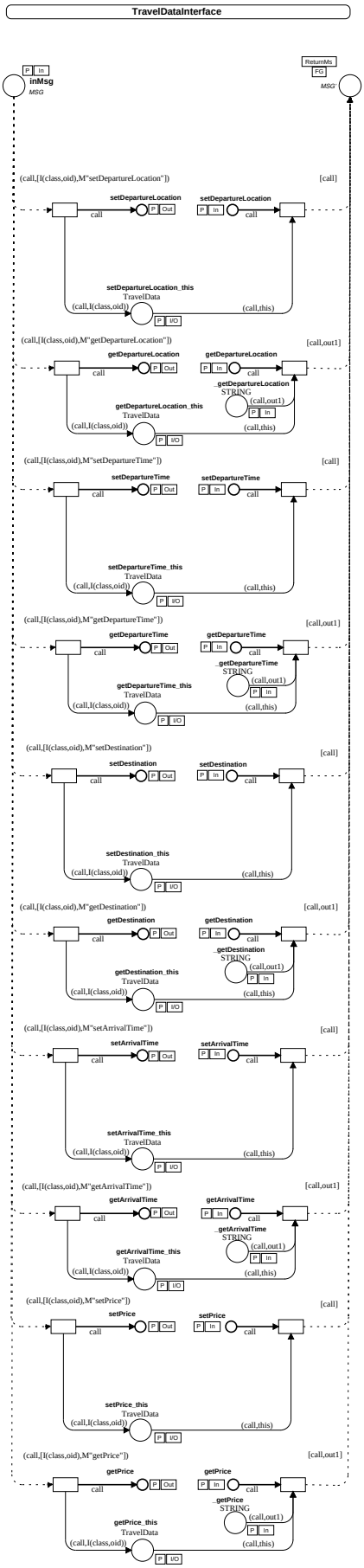

```
input (strmsg);
output (callmsgs,returnmsgs,stop);
action
 case toLocal (!my_pid) (strToMsg (strmsg)) of
    M(m)::rcv::param => (1‘(rcv::M(m)::param), empty, false)
  | I(i)::msg        => (empty,         1‘(I(i)::msg), false)
  | msg              => (empty,         1‘msg,         true);
```

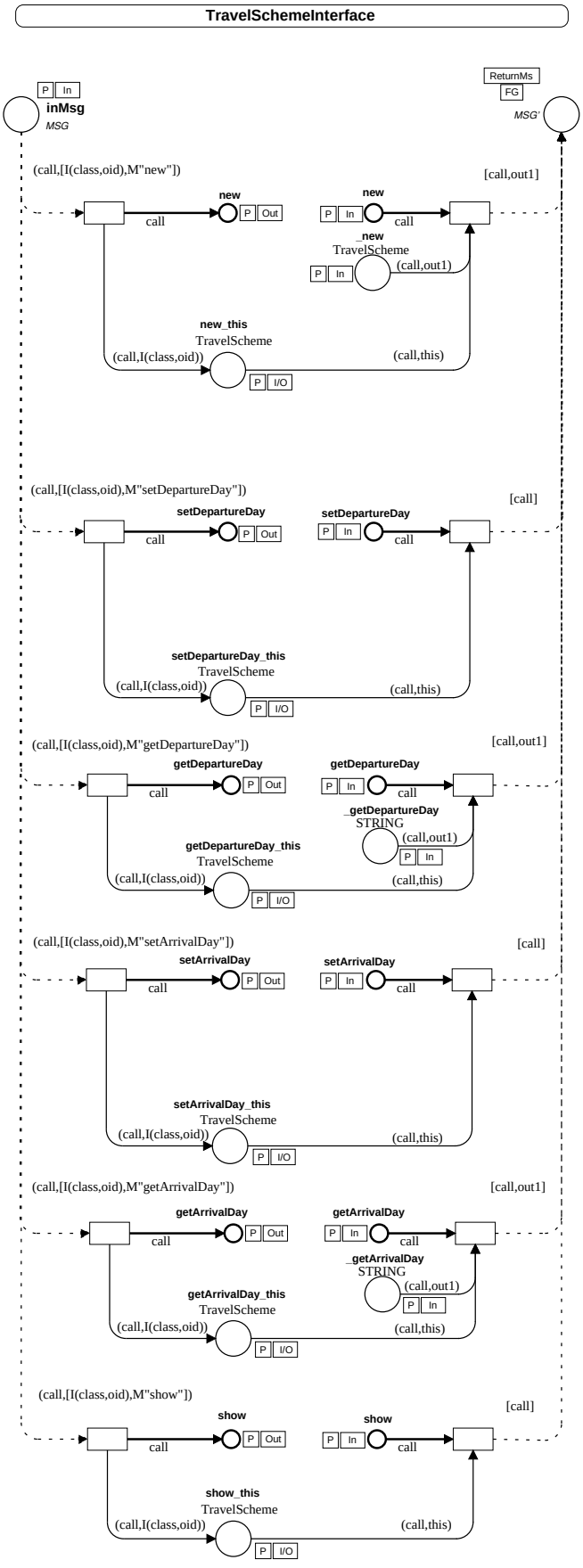

Hierarchy#10010
Remote#6
M
Prime
CodePage#1
***#0
TravelSchemeContainerInterface#39
implements
TravelSchemeContainer#15
M
Prime
***
new
createTravel
browse
TravelSchemeContainer.new()#9
TSC.createTravelScheme()#29
TSC.browse()#65
TravelScheme.new()
this.storeTravelScheme(ts)
TravelScheme.show()
Call#20
Class#5
TravelDataInterface#53
implements
TravelSchemeInterface#52
implements
TravelScheme#33
M
Prime
***
new
show
TravelScheme.new()#32
TravelScheme.show()#66
this.getDepartureLocation()
this.getDepartureDay()
this.getDepartureTime()
this.getDestination()
this.getArrivalDay()
this.getArrivalTime()
this.getPrice()
this.setDepartureLocation()
this.setDepartureDay()
this.setDepartureTime()
this.setDestination()
this.setArrivalDay()
this.setArrivalTime()
this.setPrice()

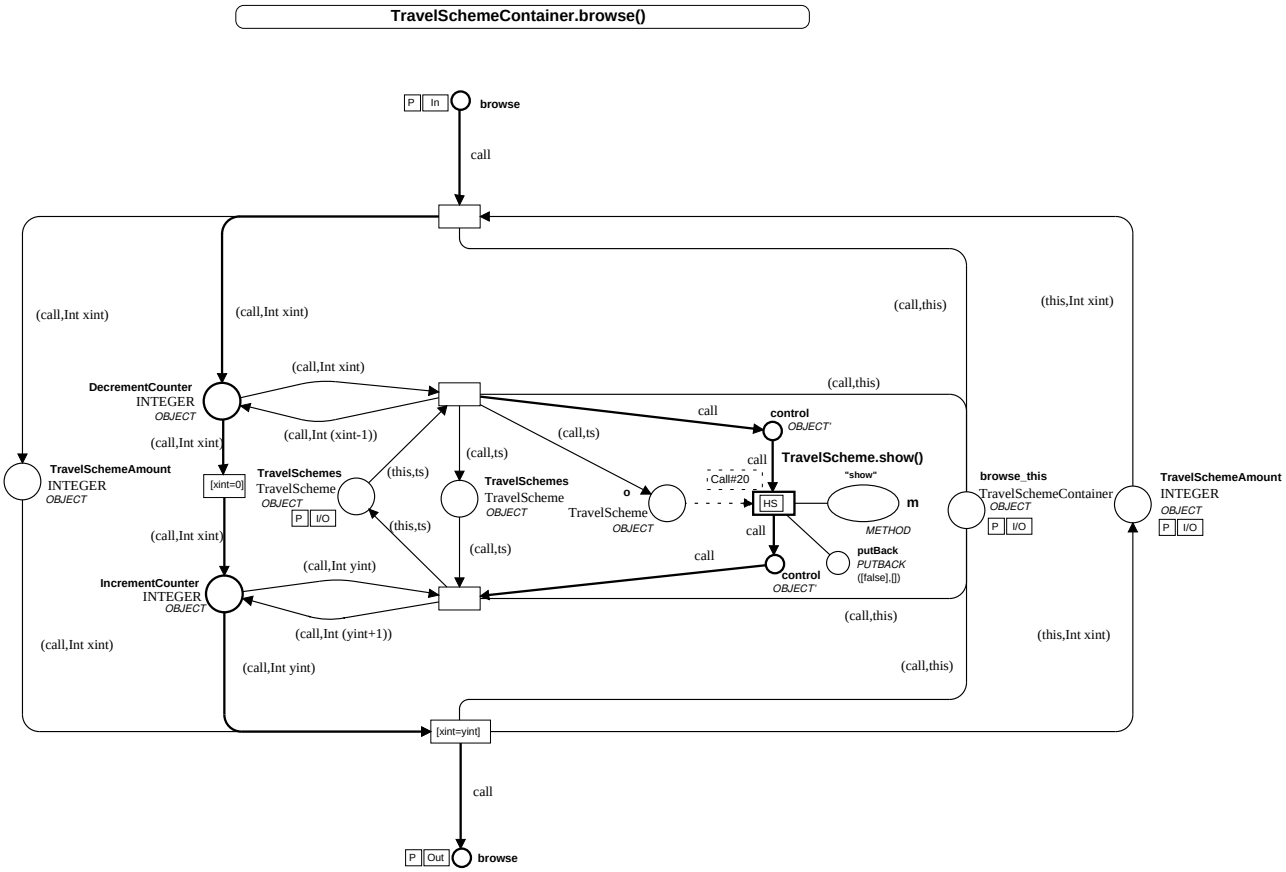

TravelScheme.show()
show_this
TravelScheme
P
I/O
show
P
In
call
fork
HS
dl
dday
dt
d
aday
at
p
"getDepartureLocation"
"getDepartureDay"
"getDepartureTime"
"getDestination"
"getArrivalDay"
"getArrivalTime"
"getPrice"
this.getDepartureLocation()
this.getDepartureDay()
this.getDepartureTime()
this.getDestination()
this.getArrivalDay()
this.getArrivalTime()
this.getPrice()
join
call
(call,Str dt)
(call,Str d)
(call,Str dday)
(call,Str aday)
call
(call,Str at)
(call,Str dl)
(call,Str p)
C
input (dl,dday,dt,d,aday,at,p);
action
DSUI_UserAckMessage("TravelScheme:
"^chr(10)^"Departure location: "^dl^""^" -
"^"Destination: "^d^""^chr(10)^"Departure
day: "^dday^""^" - "^"Departure time:
"^dt^""^chr(10)^"Arrival day: "^aday^""^" -
"^"Arrival time: "^at^""^chr(10)^"Price:
"^p^"");
call
P
Out
show


TravelScheme.new()
new_this
TravelScheme
P I/O
P In
new
call
(call,this)
(this,Str "")
init
dl
dday
dt
d
aday
at
p
P Out
call
fork
HS
"setDepartureLocation"
"setDepartureDay"
"setDepartureTime"
"setDestination"
"setArrivalDay"
"setArrivalTime"
"setPrice"
this.setDepartureLocation()
this.setDepartureDay()
this.setDepartureTime()
this.setDestination()
this.setArrivalDay()
this.setArrivalTime()
this.setPrice()
join
call
(call,this)
call
(call,this)
_new
TravelScheme
P Out
P Out
new

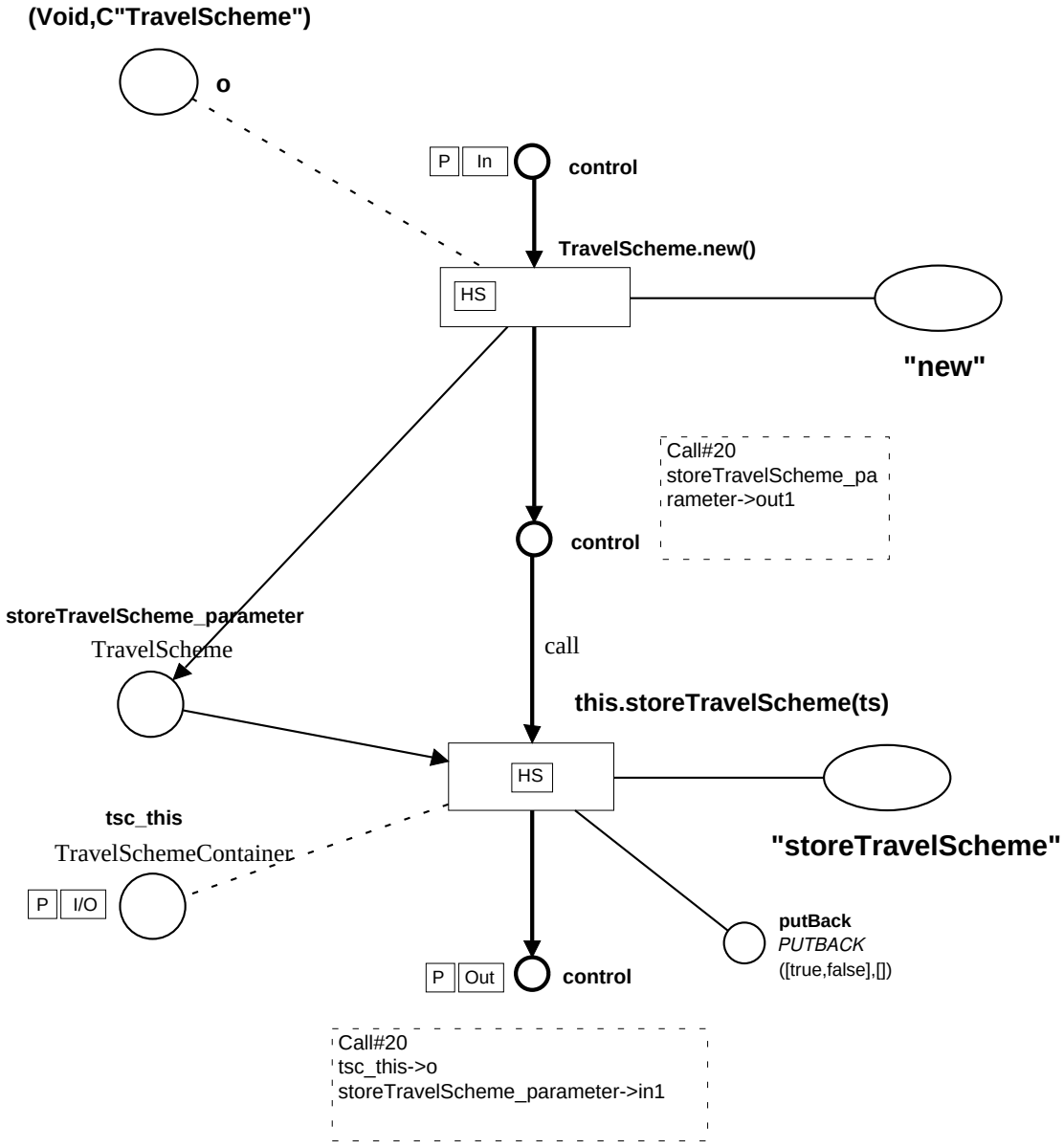

TravelScheme
"TravelScheme"
Class
inMsg
MSG
HS
Class#5
HS
TravelDataInterface#53
HS
TravelSchemeInterface#52
dl
dday
dt
d
aday
at
p
setDepartureLocation
call
setDeparture Location
C
(call,this)
(this,Str dl)
(this,Str instring)
dl
STRING
setDepartureLocation_this
TravelData
OBJECT
(this,Str dl)
getDepartureLocation
getDeparture Location
(call,Str dl)
getDepartureLocation STRING
getDepartureLocation_this
TravelData
input instring;
output dl;
action
GetString("DepartureLocation: ","%s");
new
new
HS
TravelScheme.new()
new_this
TravelScheme
setDepartureDay
setDeparture Day
(this,Str dday)
dday
setDepartureDay_this
TravelScheme
input instring;
output dday;
action
GetString("DepartureDay: ","%s");
getDepartureDay
getDeparture Day
getDepartureDay STRING
getDepartureDay_this
TravelScheme
(call,Str dday)
setDepartureTime
setDeparture Time
(this,Str dt)
dt
setDepartureTime_this
TravelData
(this,Str dt)
input instring;
output dt;
action
GetString("DepartureTime: ","%s");
getDepartureTime
getDeparture Time
(call,Str dt)
getDepartureTime STRING
getDepartureTime_this
TravelData
setArrivalDay
setArrival Day
(this,Str aday)
aday
setArrivalDay_this
TravelScheme
(this,Str aday)
input instring;
output aday;
action
GetString("ArrivalDay: ","%s");
getArrivalDay
getArrivalDay
getArrivalDay STRING
getArrivalDay_this
TravelScheme
(call,Str aday)
setDestination
setDestination
(this,Str d)
d
setDestination_this
TravelData
(this,Str d)
input instring;
output d;
action
GetString("Destination: ","%s");
getDestination
getDestination
getDestination STRING
getDestination_this
TravelData
(call,Str d)
show
show
HS
TravelScheme.show()#66
show_this
TravelScheme
setArrivalTime
setArrivalTime
(this,Str at)
at
setArrivalTime_this
TravelData
(this,Str at)
input instring;
output at;
action
GetString("ArrivalTime: ","%s");
getArrivalTime
getArrivalTime
getArrivalTime STRING
getArrivalTime_this
TravelData
(call,Str at)
setPrice
setPrice
(this,Str p)
p
setPrice_this
TravelData
(this,Str p)
input instring;
output p;
action
GetString("Price: ","%s");
getPrice
getPrice
getPrice STRING
getPrice_this
TravelData
(call,Str p)


TravelDataInterface
P In
inMsg
MSG
ReturnMs
FG
MSG
(call,[I(class,oid),M"setDepartureLocation"])
[call]
setDepartureLocation
call
P Out
setDepartureLocation
P In
call
setDepartureLocation_this
TravelData
(call,I(class,oid))
P I/O
(call,this)
(call,[I(class,oid),M"getDepartureLocation"])
[call,out1]
getDepartureLocation
call
P Out
getDepartureLocation
P In
call
_getDepartureLocation
STRING
(call,out1)
P In
getDepartureLocation_this
TravelData
(call,I(class,oid))
P I/O
(call,this)
(call,[I(class,oid),M"setDepartureTime"])
[call]
setDepartureTime
call
P Out
setDepartureTime
P In
call
setDepartureTime_this
TravelData
(call,I(class,oid))
P I/O
(call,this)
(call,[I(class,oid),M"getDepartureTime"])
[call,out1]
getDepartureTime
call
P Out
getDepartureTime
P In
call
_getDepartureTime
STRING
(call,out1)
P In
getDepartureTime_this
TravelData
(call,I(class,oid))
P I/O
(call,this)
(call,[I(class,oid),M"setDestination"])
[call]
setDestination
call
P Out
setDestination
P In
call
setDestination_this
TravelData
(call,I(class,oid))
P I/O
(call,this)
(call,[I(class,oid),M"getDestination"])
[call,out1]
getDestination
call
P Out
getDestination
P In
call
_getDestination
STRING
(call,out1)
P In
getDestination_this
TravelData
(call,I(class,oid))
P I/O
(call,this)
(call,[I(class,oid),M"setArrivalTime"])
[call]
setArrivalTime
call
P Out
setArrivalTime
P In
call
setArrivalTime_this
TravelData
(call,I(class,oid))
P I/O
(call,this)
(call,[I(class,oid),M"getArrivalTime"])
[call,out1]
getArrivalTime
call
P Out
getArrivalTime
P In
call
_getArrivalTime
STRING
(call,out1)
P In
getArrivalTime_this
TravelData
(call,I(class,oid))
P I/O
(call,this)
(call,[I(class,oid),M"setPrice"])
[call]
setPrice
call
P Out
setPrice
P In
call
setPrice_this
TravelData
(call,I(class,oid))
P I/O
(call,this)
(call,[I(class,oid),M"getPrice"])
[call,out1]
getPrice
call
P Out
getPrice
P In
call
_getPrice
STRING
(call,out1)
P In
getPrice_this
TravelData
(call,I(class,oid))
P I/O
(call,this)

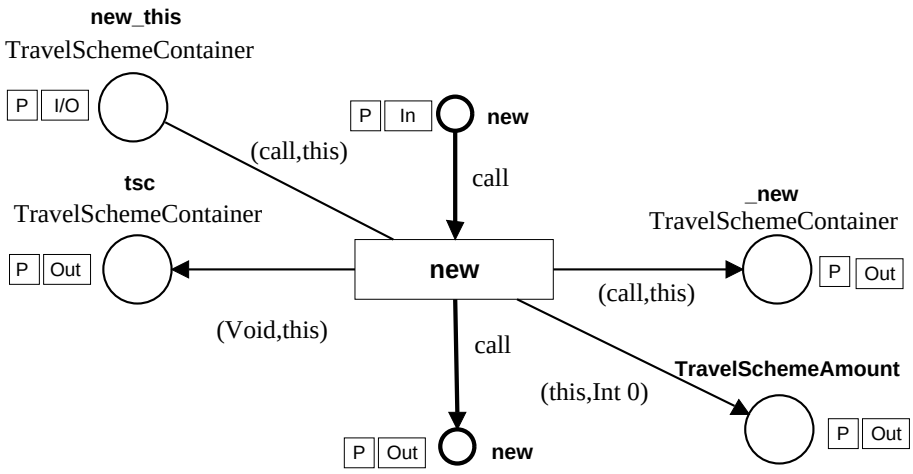

TravelSchemeInterface
P In
inMsg
MSG
ReturnMs
FG
MSG'
(call,[I(class,oid),M"new"])
[call,out1]
new
P Out
call
new
P In
call
_new
TravelScheme
P In
(call,out1)
new_this
TravelScheme
(call,I(class,oid))
(call,this)
P I/O
(call,[I(class,oid),M"setDepartureDay"])
[call]
setDepartureDay
P Out
call
setDepartureDay
P In
call
setDepartureDay_this
TravelScheme
(call,I(class,oid))
(call,this)
P I/O
(call,[I(class,oid),M"getDepartureDay"])
[call,out1]
getDepartureDay
P Out
call
getDepartureDay
P In
call
_getDepartureDay
STRING
(call,out1)
P In
getDepartureDay_this
TravelScheme
(call,I(class,oid))
(call,this)
P I/O
(call,[I(class,oid),M"setArrivalDay"])
[call]
setArrivalDay
P Out
call
setArrivalDay
P In
call
setArrivalDay_this
TravelScheme
(call,I(class,oid))
(call,this)
P I/O
(call,[I(class,oid),M"getArrivalDay"])
[call,out1]
getArrivalDay
P Out
call
getArrivalDay
P In
call
_getArrivalDay
STRING
(call,out1)
P In
getArrivalDay_this
TravelScheme
(call,I(class,oid))
(call,this)
P I/O
(call,[I(class,oid),M"show"])
[call]
show
P Out
call
show
P In
call
show_this
TravelScheme
(call,I(class,oid))
(call,this)
P I/O


TravelSchemeContainer.browse()
P In
browse
call
(call,Int xint)
(call,Int xint)
(call,this)
(this,Int xint)
DecrementCounter
INTEGER
OBJECT
(call,Int xint)
(call,Int (xint-1))
(call,ts)
(call,this)
call
control
OBJECT'
TravelScheme.show()
"show"
m
METHOD
Call#20
HS
call
putBack
PUTBACK
([false].[])
control
OBJECT'
TravelSchemeAmount
INTEGER
OBJECT
[xint=0]
TravelSchemes
TravelScheme
OBJECT
P I/O
(this,ts)
(call,ts)
TravelSchemes
TravelScheme
OBJECT
o
TravelScheme
OBJECT
(this,ts)
(call,ts)
(call,Int xint)
(call,Int yint)
IncrementCounter
INTEGER
OBJECT
(call,Int (yint+1))
(call,this)
(call,Int xint)
(call,Int yint)
(call,this)
(this,Int xint)
browse_this
TravelSchemeContainer
OBJECT
P I/O
TravelSchemeAmount
INTEGER
OBJECT
P I/O
[xint=yint]
call
P Out
browse

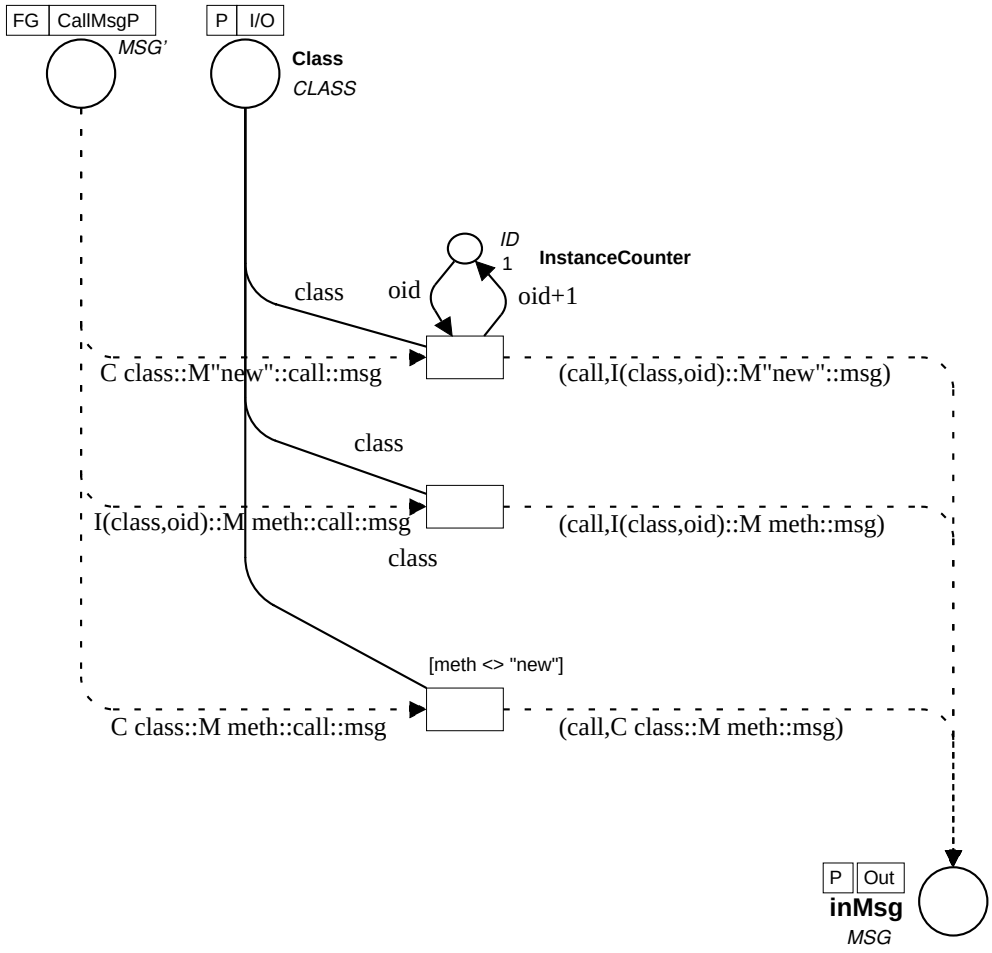
TravelSchemeContainer.createTravelScheme()



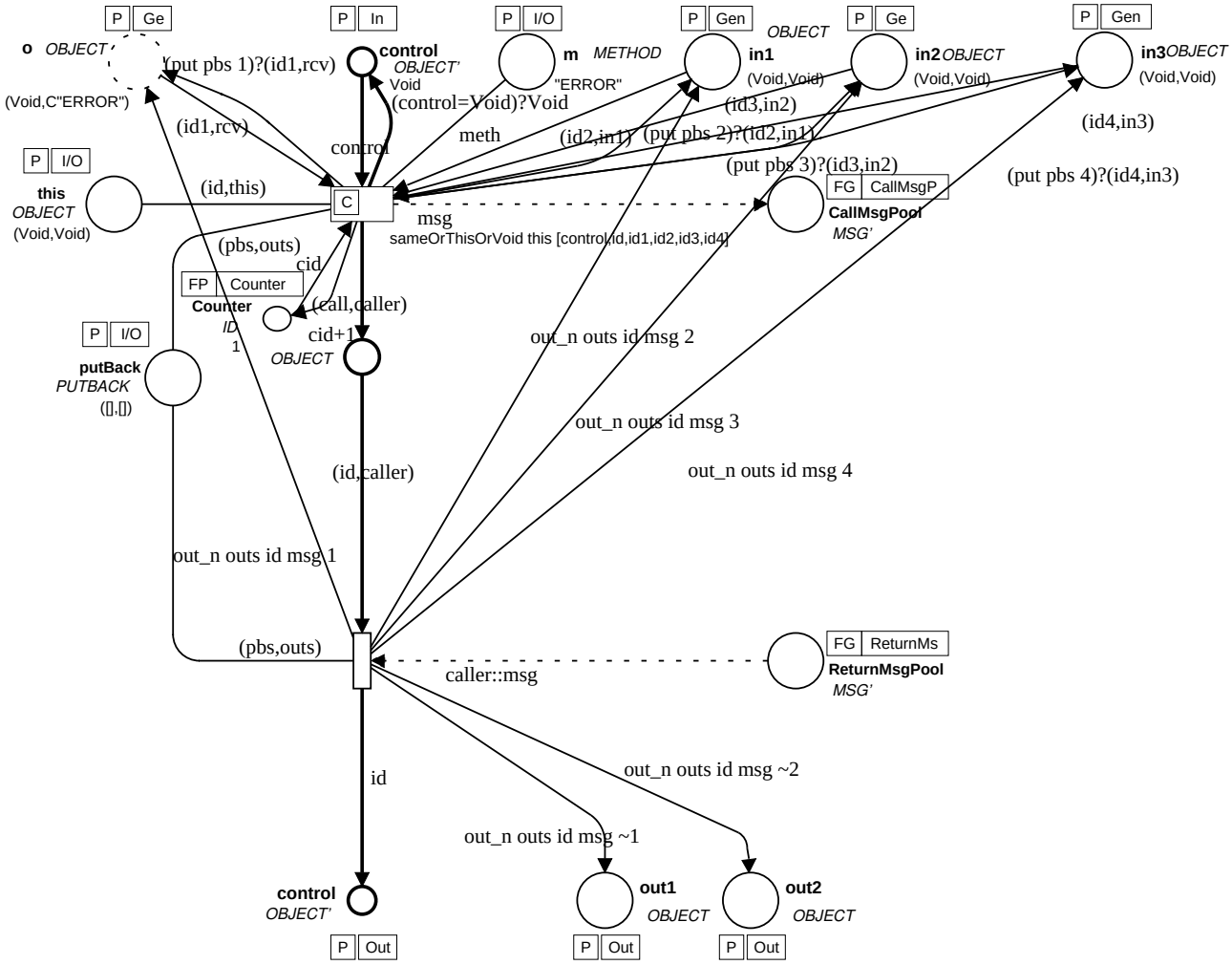
TravelSchemeContainer.new()



Class-Page


FG
CallMsgP
MSG'
P
I/O
Class
CLASS
ID
1
InstanceCounter
class
oid
oid+1
C class::M"new"::call::msg
(call,I(class,oid)::M"new"::msg)
class
I(class,oid)::M meth::call::msg
(call,I(class,oid)::M meth::msg)
class
[meth <> "new"]
C class::M meth::call::msg
(call,C class::M meth::msg)
P
Out
inMsg
MSG

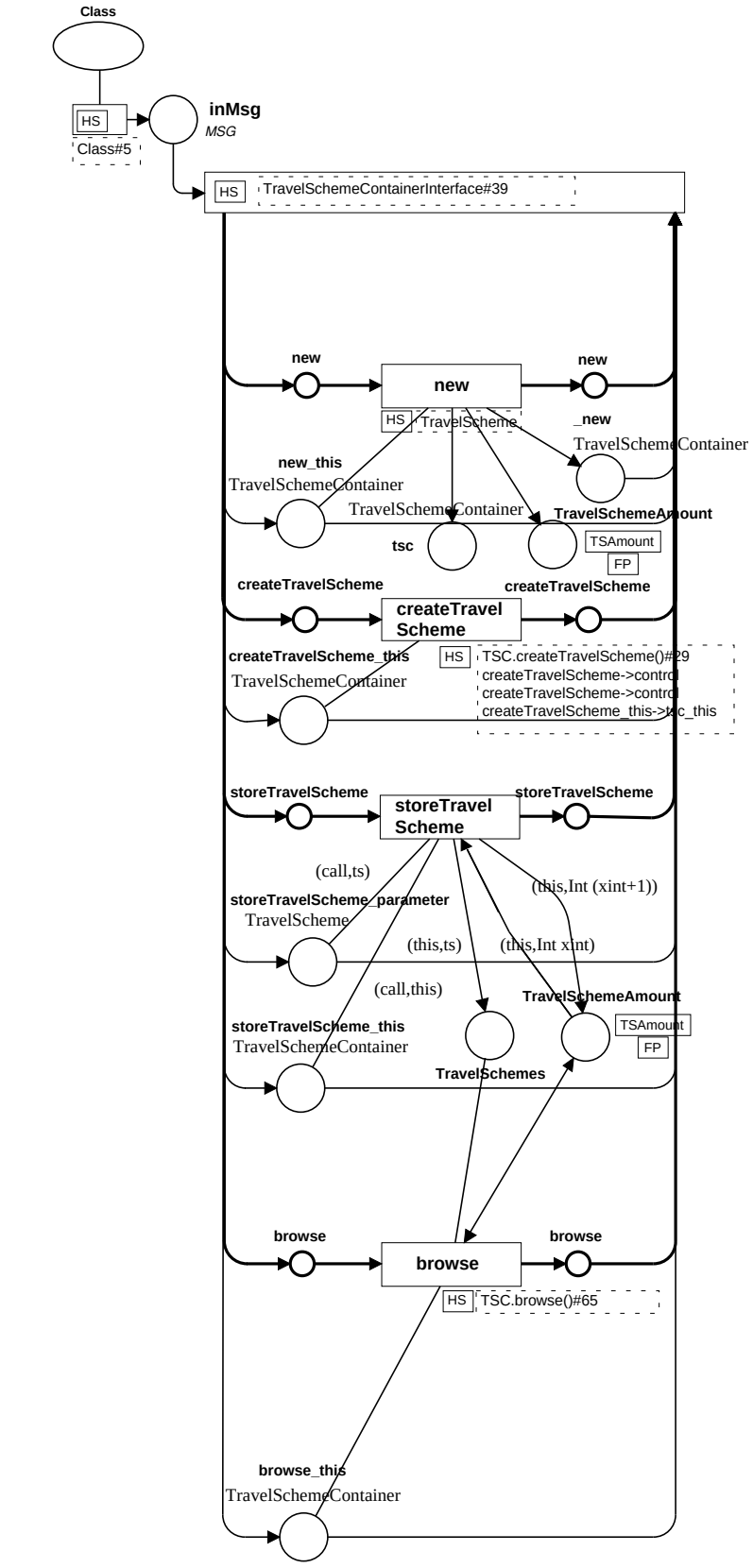
Call-Page

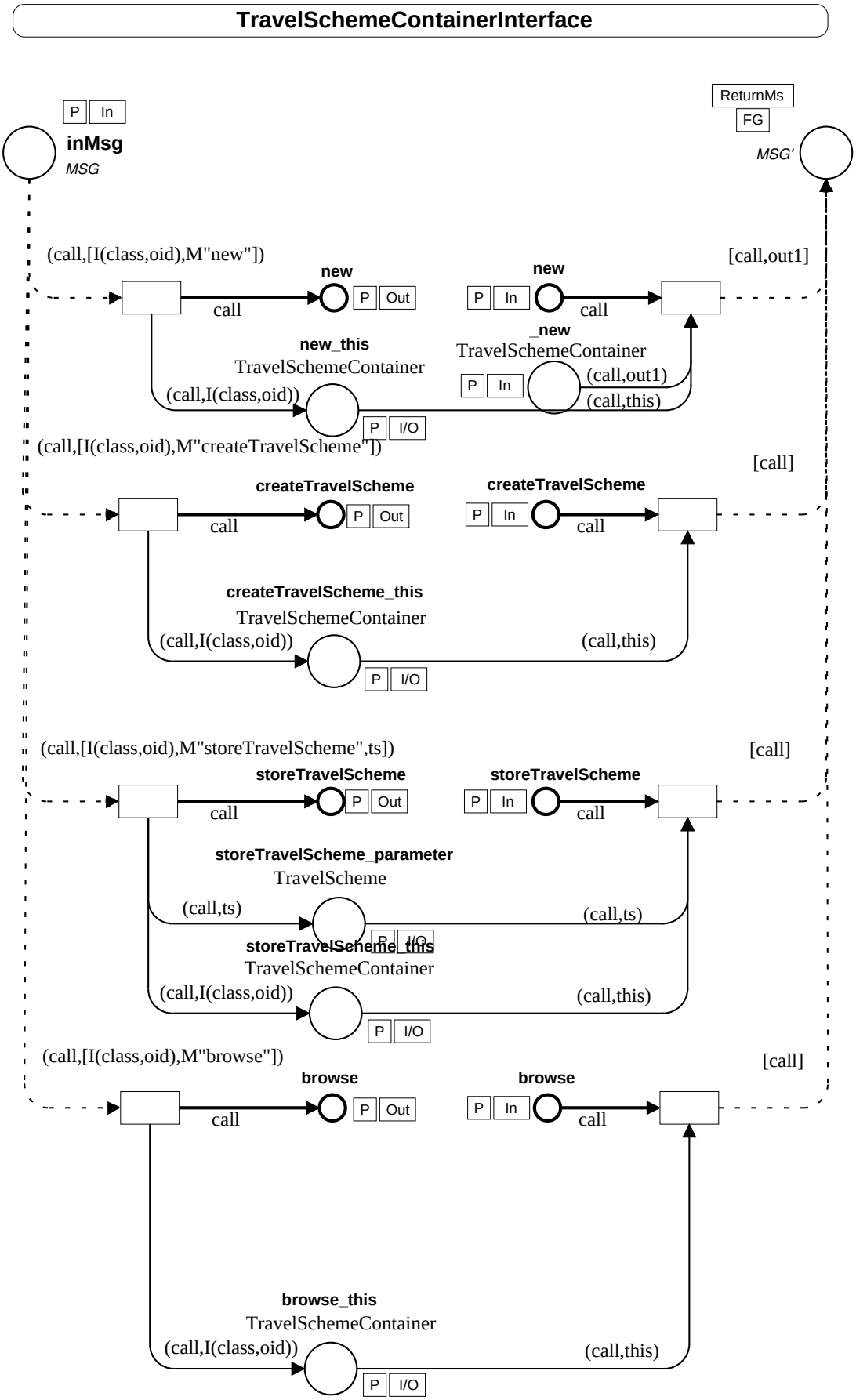



```
input (control,id,this,id1,rcv,id2,meth,id3,in1,in2,cid,in3,id4);
output(call,caller,msg);
action
  let val cntrl = nonThisOrVoid this [control,id,id1,id2,id3,id4];
    val caller = I("Call",cid);
in (cntrl,caller,rcv::M meth::caller::(noVoid [in1,in2,in3]))
end;
```

TravelSchemeContainer

"TravelSchemeContainer"




TravelSchemeContainerInterface
P
In
inMsg
MSG
ReturnMs
FG
MSG'
(call,[I(class,oid),M"new"])
new
call
P
Out
new
P
In
call
[call,out1]
new_this
TravelSchemeContainer
_new
TravelSchemeContainer
(call,I(class,oid))
(call,out1)
(call,this)
P
I/O
(call,[I(class,oid),M"createTravelScheme"])
[call]
createTravelScheme
createTravelScheme_this
TravelSchemeContainer
(call,I(class,oid))
(call,this)
(call,[I(class,oid),M"storeTravelScheme",ts])
storeTravelScheme
storeTravelScheme_parameter
TravelScheme
(call,ts)
storeTravelScheme_this
(call,[I(class,oid),M"browse"])
browse
browse_this

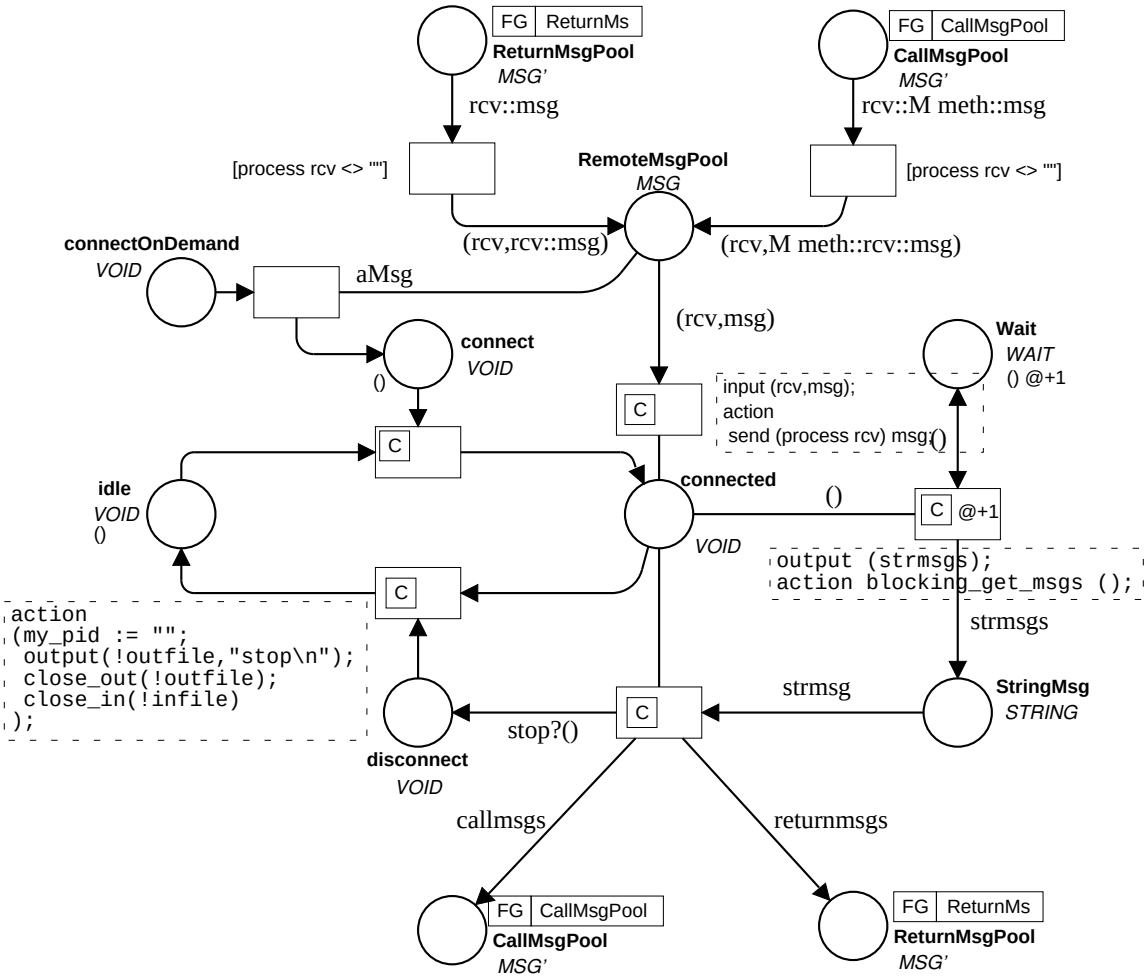
Code-Page

```
(* workaround: *)
val my_execute = execute;

color WAIT = unit timed;
color VOID = unit;
color INTEGER = int;
color REAL = real;
color STRING = string;
color BOOLEAN = bool;
color ID = INTEGER declare input_col;
color RID = STRING;
color BOOLEANLIST = list BOOLEAN;
color INTEGERLIST = list INTEGER;
color PUTBACK = product BOOLEANLIST * INTEGERLIST;
color VALUE = with Integer | RealNumber | String | Boolean;
color MODIFIER = with Static;
color MODIFIERS = list MODIFIER;
color CLASS = STRING;
color METHOD = STRING;
color INSTANCE = product CLASS * ID;
color RINSTANCE = product CLASS * RID * RID;
color RCLASS = product CLASS * RID;
color OBJECT' = union Void + Null
    + C:CLASS + RC:RCLASS
    + M:METHOD
    + I:INSTANCE + RI:RINSTANCE
    + Int:INTEGER + Str:STRING + Bool:BOOLEAN + Real:REAL
    declare mkst_col, of_M;
color MSG' = list OBJECT' declare input_col;
color TYPE = union Object + Reference + ClassRef + InstRef + Value
    + Method + AnyRef + Bottom
    + TC:CLASS + TI:CLASS + TM:METHOD + TV:VALUE;
color TYPELIST = list TYPE;
color MSGTYPE = product TYPELIST * TYPELIST;
color MSGDEF = product MODIFIERS * TYPELIST * METHOD * TYPELIST;
color RETURN' = product OBJECT' * TYPELIST;
color RETURN = product OBJECT'  * RETURN';
color MSG = product OBJECT'  * MSG';
color OBJECTLIST = MSG;
color OBJECT = product OBJECT'  * OBJECT';
color RIDxMSG' = product RID * MSG';

fun GetString(cr_prompt:string,cr_def:string,canc:string)=
let val Stringvalue =
DSUI_GetString{prompt=cr_prompt,def=cr_def}
in
  if Stringvalue <> "" then
       Stringvalue
  else
     GetString(cr_prompt,cr_def,canc)
  end
  handle _ => canc;

fun GetInteger(cr_prompt:string,cr_def:int,canc:int)=
let val Integervalue =
DSUI_GetIntegerValue{prompt=cr_prompt,def=cr_def}
in
  if Integervalue > 0 then
       Integervalue
  else
     GetInteger(cr_prompt,cr_def,canc)
  end
  handle _ => canc;



(* Net Inscription Abbreviations: *)
infix 5 ?;
fun p?v = if p then 1'v else empty;

infix 8 // ;
fun x // y = (x,y);

local fun mkst_list' _ nil = ""
  | mkst_list' mkst [elem] = mkst elem
  | mkst_list' mkst (elem::lst) = mkst elem ^ "," ^ mkst_list' mkst lst
in fun mkst_list mkst lst = "[" ^ mkst_list' mkst lst ^ "]" end;

val msgToStr = mkst_list mkst_col'OBJECT';

fun strToMsg strmsg = (input_col'MSG' (open_string strmsg)) handle _ => [];

local fun remLocal' pid (C(class)) = RC(class,pid)
  | remLocal' pid (I(class,oid)) = RI(class,pid,makestring oid)
  | remLocal' _ old = old
in fun remLocal pid msg = map (remLocal' pid) msg end;

local fun toLocal' pid (old as RC(class,pid')) = if pid=pid' then C(class) else old
  | toLocal' pid (old as RI(class,pid',roid)) =
     if pid=pid' then I(class,input_col'ID (open_string roid)) else old
  | toLocal' _ old = old
in fun toLocal pid msg = map (toLocal' pid) msg end;


fun getType Void = Bottom
  | getType Null = AnyRef
  | getType (C class) = TC class
  | getType (RC(class,_)) = TC class
  | getType (M meth) = TM meth
  | getType (I(class,_)) = TI class
  | getType (RI(class,_,_)) = TI class
  | getType (Int _) = TV Integer
  | getType (Str _) = TV String
  | getType (Bool _) = TV Boolean
  | getType (Real _) = TV RealNumber;

infix 8 <== ;
fun type1 <== type2 =
 if type1 = type2 then true
 else case (type1,type2) of
   (_,          Object  ) => true
     | (Bottom, _       ) => true
     | (AnyRef, TC _    ) => true
     | (AnyRef, TI _    ) => true
     | (AnyRef, ClassRef ) => true
     | (AnyRef, InstRef  ) => true
     | (AnyRef, Reference) => true
     | (TC _  , ClassRef ) => true
     | (TC _  , Reference) => true
     | (TI _  , InstRef  ) => true
     | (TI _  , Reference) => true
     | (TM _  , Method   ) => true
     | (TV _  , Value    ) => true
     | (_     , _        ) => false;

fun ofType typ obj = getType obj <== typ;
val ofUser = ofType (TI "User");
val ofTail = ofType (TI "tail");
val ofPair = ofType (TI "Pair");


fun typeCheck nil nil = true
  | typeCheck (typ::types) (obj::objs) =
 ofType typ obj andalso typeCheck types objs
  | typeCheck _ _ = false;

fun nonThisOrVoid _ nil = Void (* darf nicht vorkommen! *)
  | nonThisOrVoid this (x::objs) = if x=Void orelse x=this then nonThisOrVoid this objs
          else x;

local
fun sameOrThisOrVoid' y _ nil = y <> Void
  | sameOrThisOrVoid' Void this (x::objs) =
  if x=Void orelse x=this then sameOrThisOrVoid' Void this objs
          else sameOrThisOrVoid' x this objs
  | sameOrThisOrVoid' y this (x::objs) =
  if x=Void orelse x=this orelse x=y then sameOrThisOrVoid' y this objs
          else false
in val sameOrThisOrVoid = sameOrThisOrVoid' Void end;

fun noVoid nil = nil
  | noVoid (Void::xs) = noVoid xs
  | noVoid (x::xs) = x::noVoid xs;

fun prefix xs 0 = nil
  | prefix (x::xs) n = x::prefix xs (n-1)
  | prefix nil _ = nil;

fun put (x::xs) 1 = x
  | put (x::xs) n = put xs (n-1)
  | put nil _ = true;

fun in_n call (obj::objs) 1 = 1'(call,obj)
  | in_n call (_::objs) n = in_n call objs (n-1)
  | in_n call nil _ = empty; (* 1'(call,Null); *)

local fun position' _ nil _ = 0
  | position' n (x::xs) y = if x = y then n else position' (n+1) xs y
in val position = position' 1
end;

fun out_n outs call msg index =
 if mem outs index
  orelse (length outs < ~index andalso length msg >= ~index)
 then let val pos = if mem outs index then position outs index
          else ~index
   in 1'(call,nth(msg,pos-1)) end
 else empty;

fun process (RC(_,pid)) = pid
  | process (RI(_,pid,_)) = pid
  | process _ = "";

globref my_pid = "";
globref infile = std_in;
globref outfile = std_out;

fun send pid msg =
 (output(!outfile, "send\n"^ pid ^"\n" ^
  msgToStr(remLocal (!my_pid) msg) ^"\n");
    flush_out(!outfile));

val callcnt = ref 0;

infix 8 \;
fun (obj\meth) params =
  let val caller = I("CALL",(callcnt:= (!callcnt)+1;!callcnt))
  in (send (process (hd (remLocal (!my_pid) [obj])))
           (M meth::obj::caller::params);
           caller)
  end;

infix 8 return;
fun (obj return params) =
  (send (process (hd (remLocal (!my_pid) [obj])))
        (obj::params));

fun blocking_get_msgs () =
    (1,input_line(!infile)) !!
   (if can_input(!infile)>0
 then blocking_get_msgs ()
 else empty);
```

```
var localmsgs, callmsgs, returnmsgs: MSG' ms;
var strmsg: STRING;
var strmsgs: STRING ms;
var timer, timer': INTEGER;
var pid,roid: RID;
var pbs: BOOLEANLIST;
var outs: INTEGERLIST;
var msg: MSG';
var aMsg: MSG;
var onDemand, stop: BOOLEAN;
var class: CLASS;
var msgType: MSGTYPE;
var inClasses, outClasses: TYPELIST;
var cid,oid: ID;
var control,call,caller,
    id, id', id1, id2, id3, id4: OBJECT';
var meth: METHOD;
var obj, in1,in2,in3,out1,out2: OBJECT';
var this,classobj,snd,rcv,
 lst,pair,cons,tail,add,
    filter,filteredList,
 mapObj,filterObj: OBJECT';
var newHead: OBJECT';
var head,car,cdr,loginstring, parameter, parameter1, parameter2: OBJECT';
var isEmpty, isString, toString, result:OBJECT';
var this'cons,this'cons',
 this'left,this'left',
 this'right,this'right': OBJECT';

var x,y,tsc,btc,bc,bus,ts,bt,busID,cl:OBJECT';
var instring,choicestring,dday,aday,dl,dt,d,af,p,adate,ddate:STRING;
var inint,xint,yint,zint,tSeats,aSeats,eSeats,busNo:INTEGER;
var object, choice, loginout:OBJECT;
```

Remote-Page

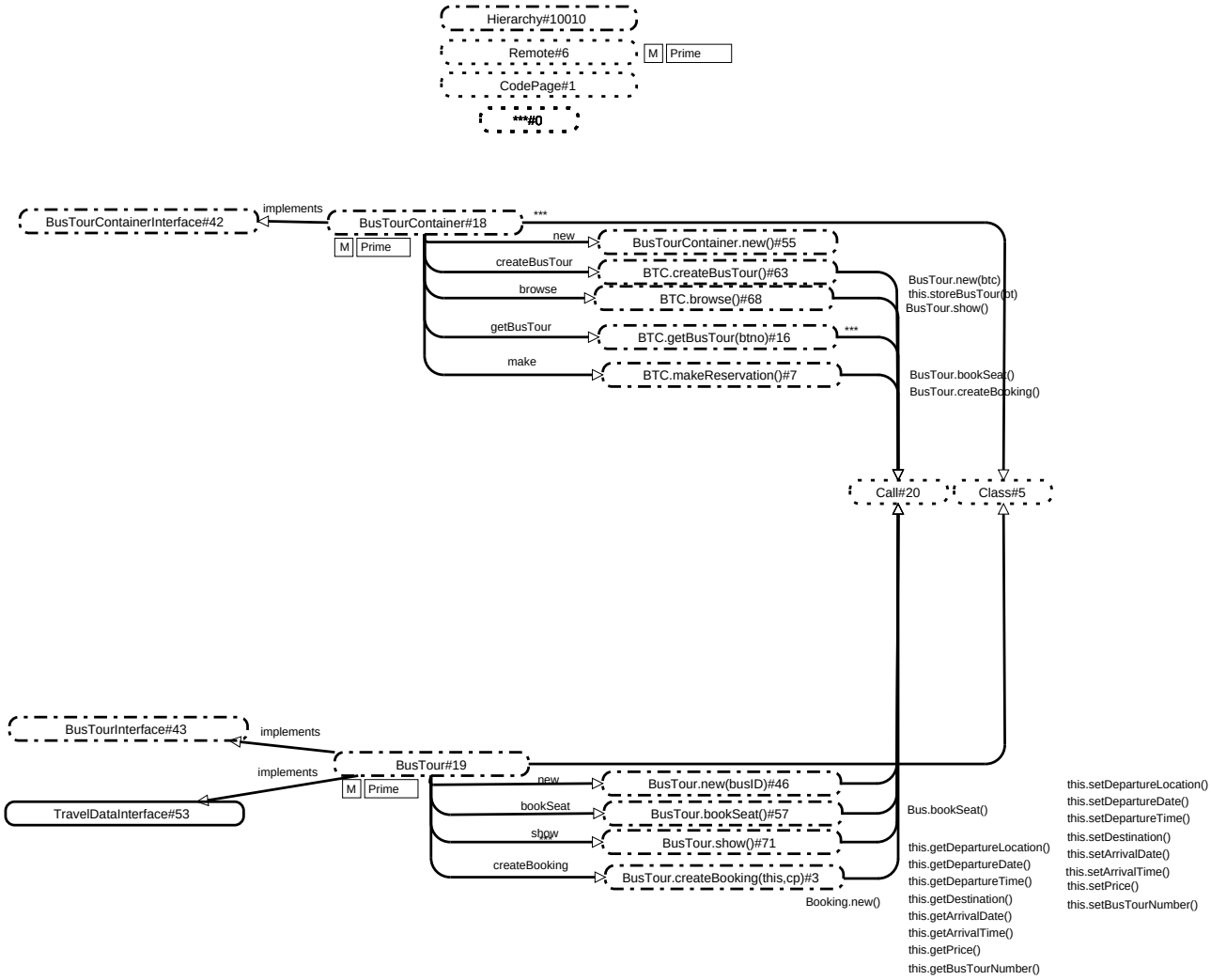
```
action
 let val name = "TravelSchemeProcess";
  val (inf,outf) = my_execute("/home/tgi_3/tgi3/bin/messenger",[name]);
 in (my_pid := name; infile := inf; outfile := outf) end;
```

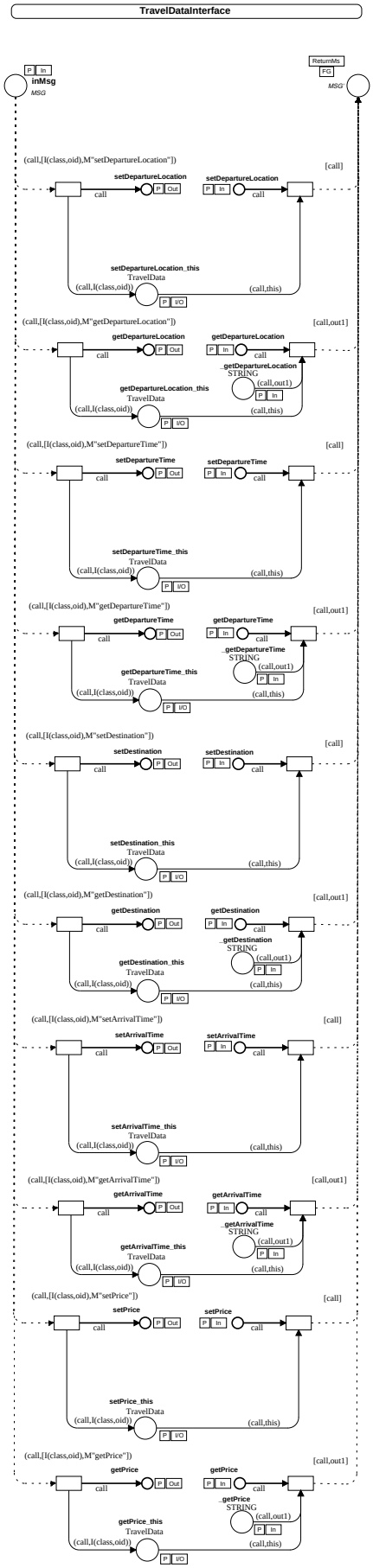



```
input (strmsg);
output (callmsgs,returnmsgs,stop);
action
 case toLocal (!my_pid) (strToMsg (strmsg)) of
    M(m)::rcv::param => (1‘(rcv::M(m)::param), empty, false)
  | I(i)::msg        => (empty,         1‘(I(i)::msg), false)
  | msg              => (empty,         1‘msg,         true);
```

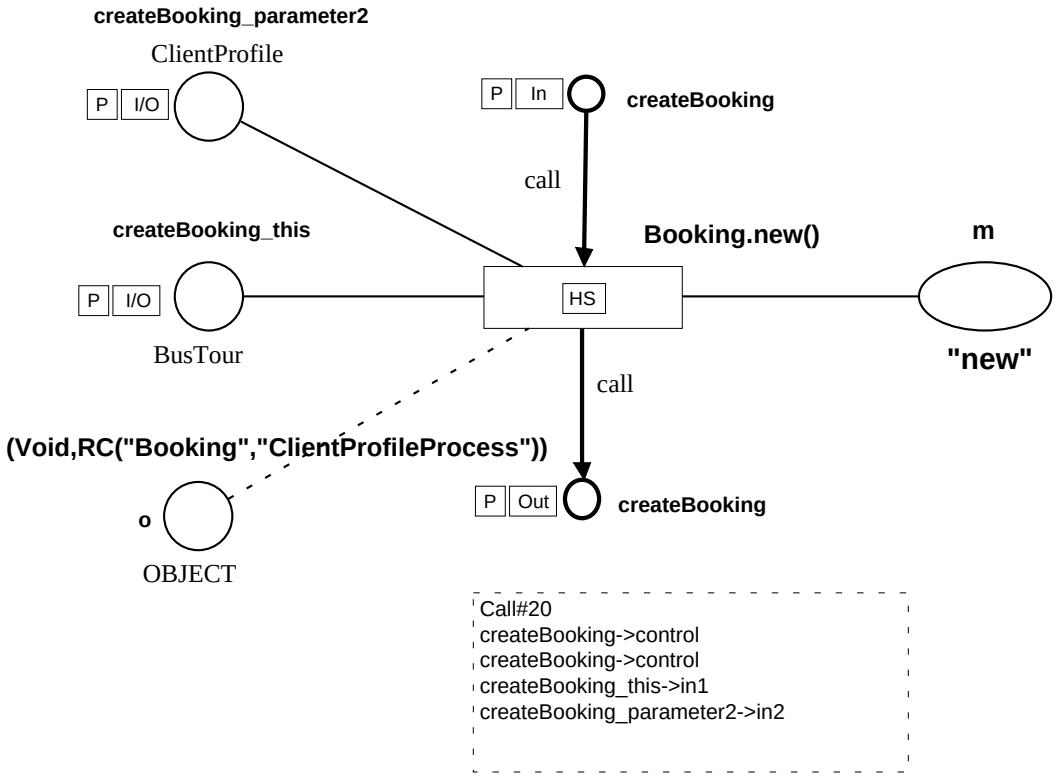
```
<<No errors>>
```

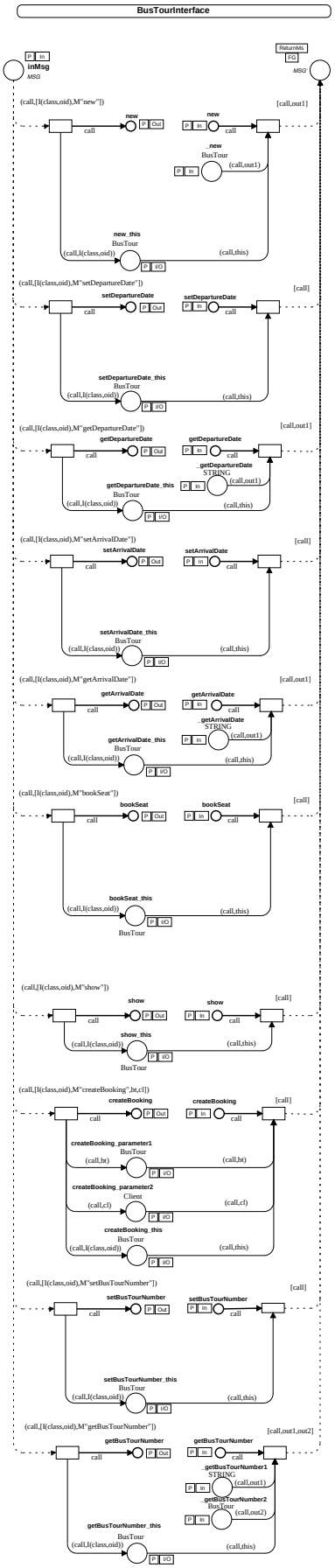
```
val it = fn : string -> OBJECT' list -> unit
```

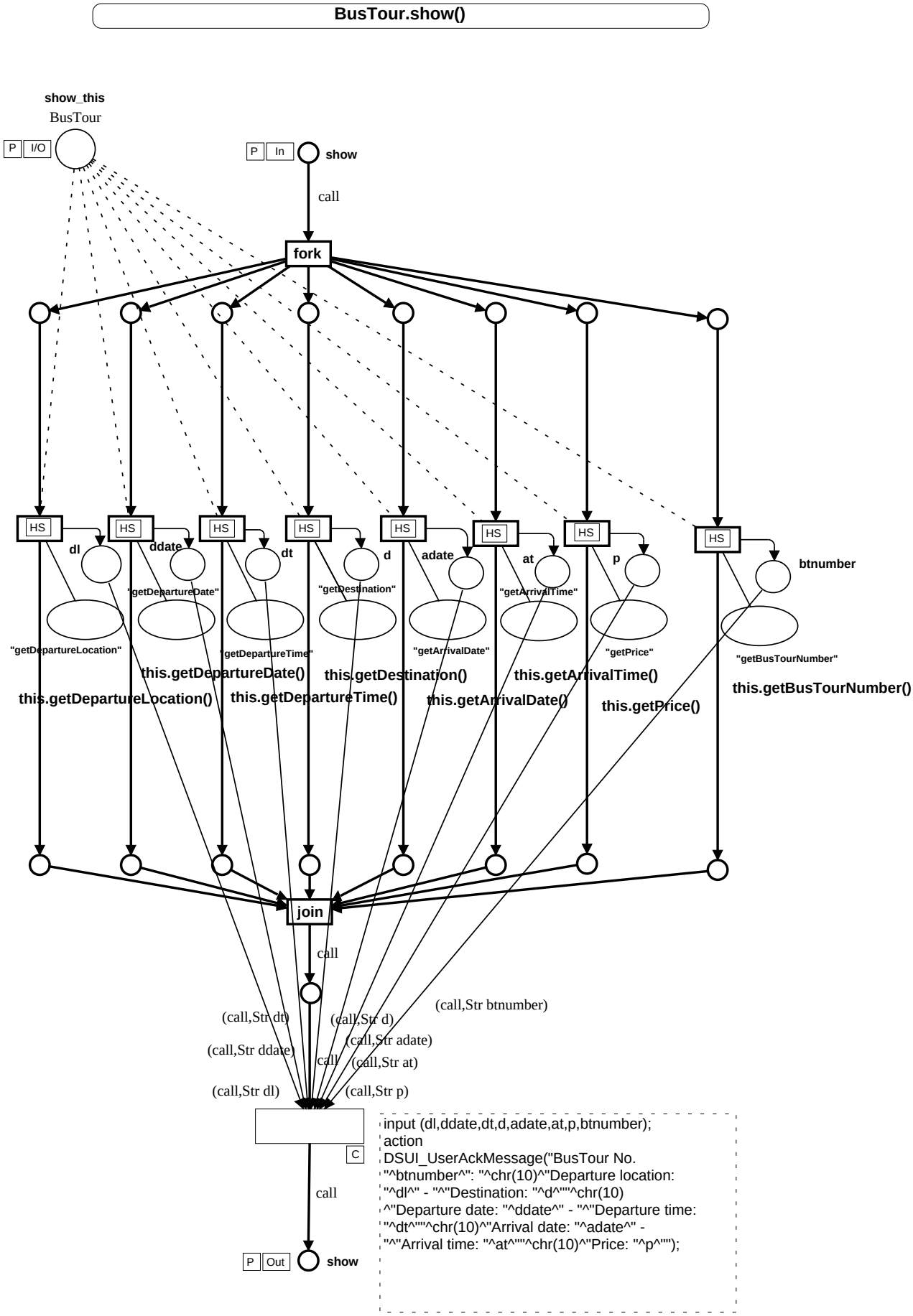

Hierarchy#10010
Remote#6
M
Prime
CodePage#1
***#0
BusTourContainerInterface#42
implements
BusTourContainer#18
***
M
Prime
new
BusTourContainer.new()#55
createBusTour
BTC.createBusTour()#63
browse
BTC.browse()#68
getBusTour
BTC.getBusTour(btno)#16
make
BTC.makeReservation()#7
BusTour.new(btc)
this.storeBusTour(bt)
BusTour.show()
***
BusTour.bookSeat()
BusTour.createBooking()
Call#20
Class#5
BusTourInterface#43
implements
BusTour#19
implements
TravelDataInterface#53
M
Prime
new
BusTour.new(busID)#46
bookSeat
BusTour.bookSeat()#57
show
BusTour.show()#71
createBooking
BusTour.createBooking(this,cp)#3
Booking.new()
Bus.bookSeat()
this.getDepartureLocation()
this.getDepartureDate()
this.getDepartureTime()
this.getDestination()
this.getArrivalDate()
this.getArrivalTime()
this.getPrice()
this.getBusTourNumber()
this.setDepartureLocation()
this.setDepartureDate()
this.setDepartureTime()
this.setDestination()
this.setArrivalDate()
this.setArrivalTime()
this.setPrice()
this.setBusTourNumber()

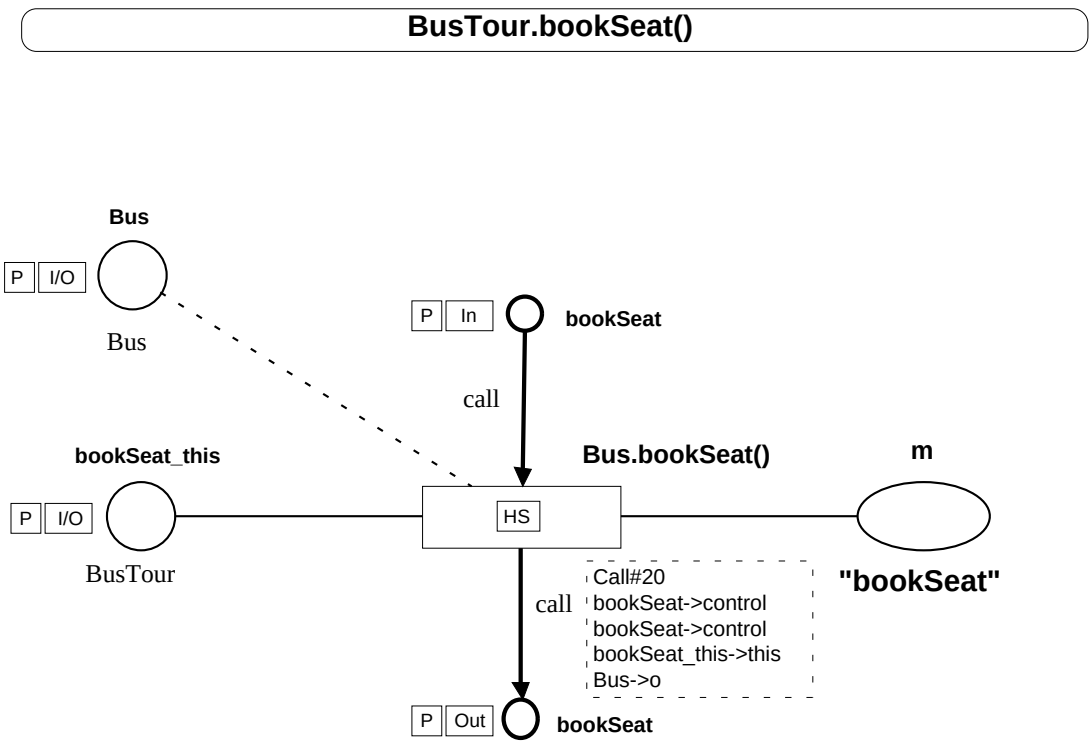

TravelDataInterface
inMsg
MSG
ReturnMs
MSG'
(call,[I(class,oid),M"setDepartureLocation"])
[call]
setDepartureLocation
call
setDepartureLocation_this
TravelData
(call,I(class,oid))
(call,this)
(call,[I(class,oid),M"getDepartureLocation"])
[call,out1]
getDepartureLocation
_getDepartureLocation
STRING
(call,out1)
getDepartureLocation_this
(call,[I(class,oid),M"setDepartureTime"])
setDepartureTime
setDepartureTime_this
(call,[I(class,oid),M"getDepartureTime"])
getDepartureTime
_getDepartureTime
getDepartureTime_this
(call,[I(class,oid),M"setDestination"])
setDestination
setDestination_this
(call,[I(class,oid),M"getDestination"])
getDestination
_getDestination
getDestination_this
(call,[I(class,oid),M"setArrivalTime"])
setArrivalTime
setArrivalTime_this
(call,[I(class,oid),M"getArrivalTime"])
getArrivalTime
_getArrivalTime
getArrivalTime_this
(call,[I(class,oid),M"setPrice"])
setPrice
setPrice_this
(call,[I(class,oid),M"getPrice"])
getPrice
_getPrice
getPrice_this

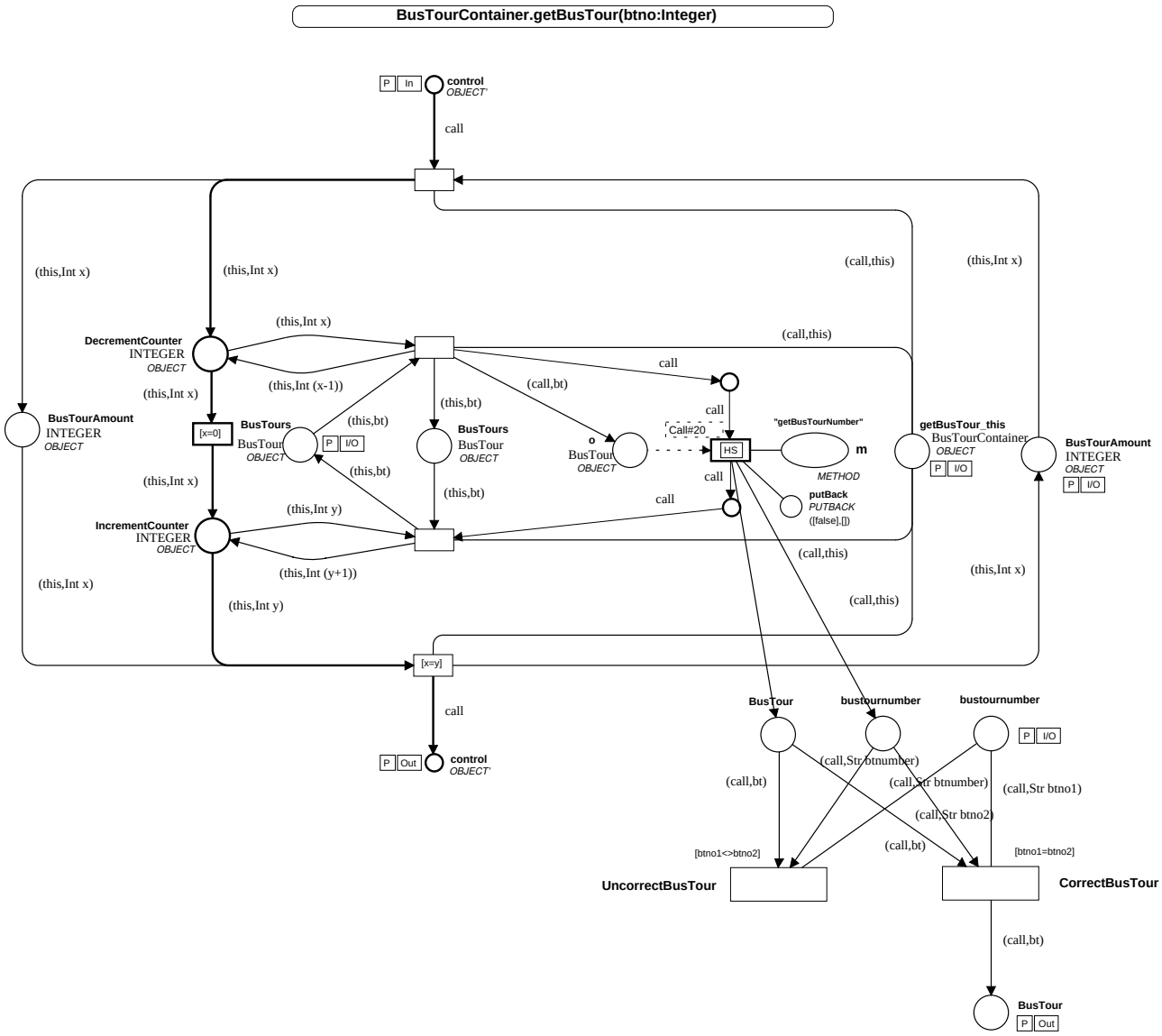
BusTour.createBooking(this:BusTour,cp:ClientProfile)

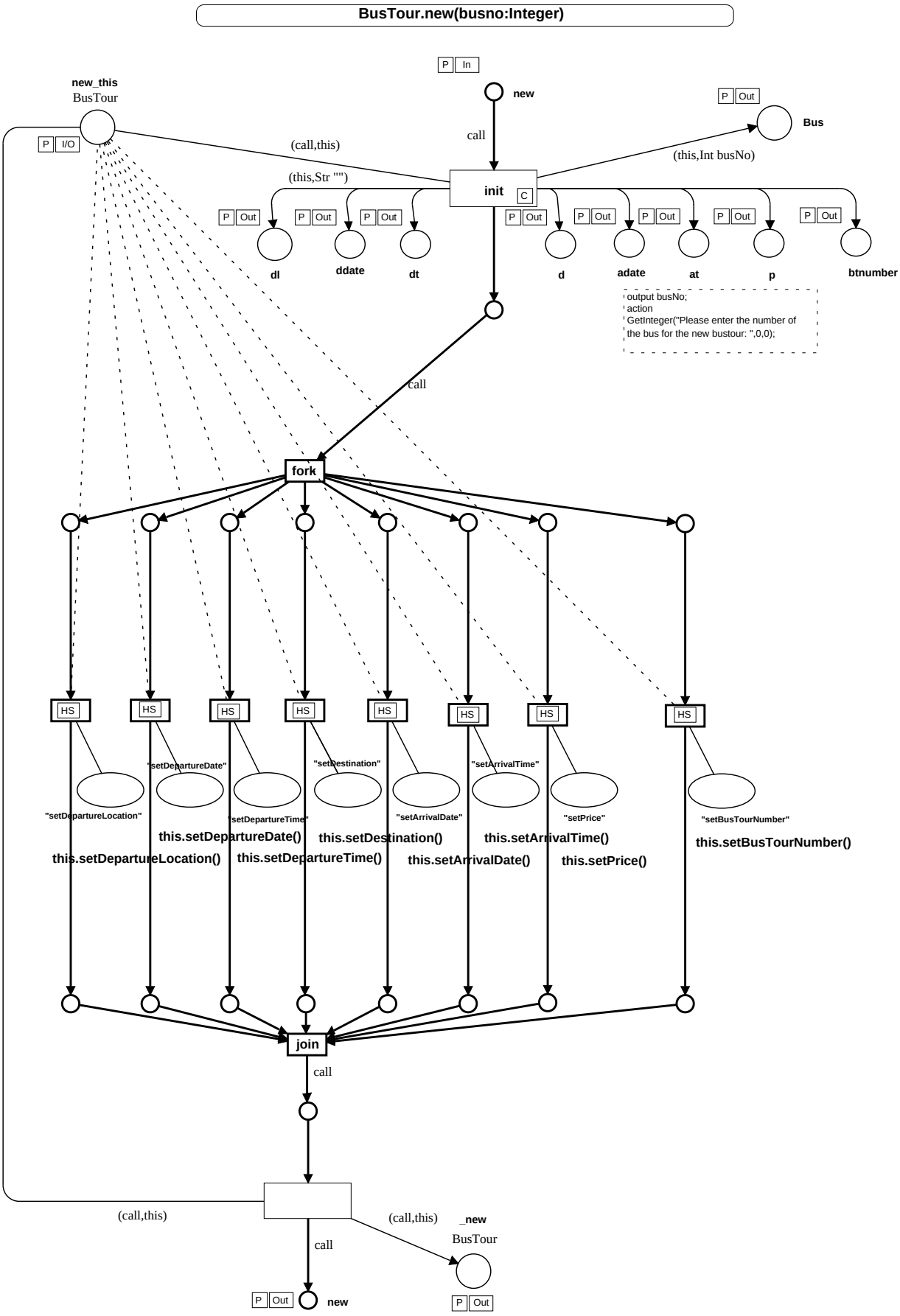


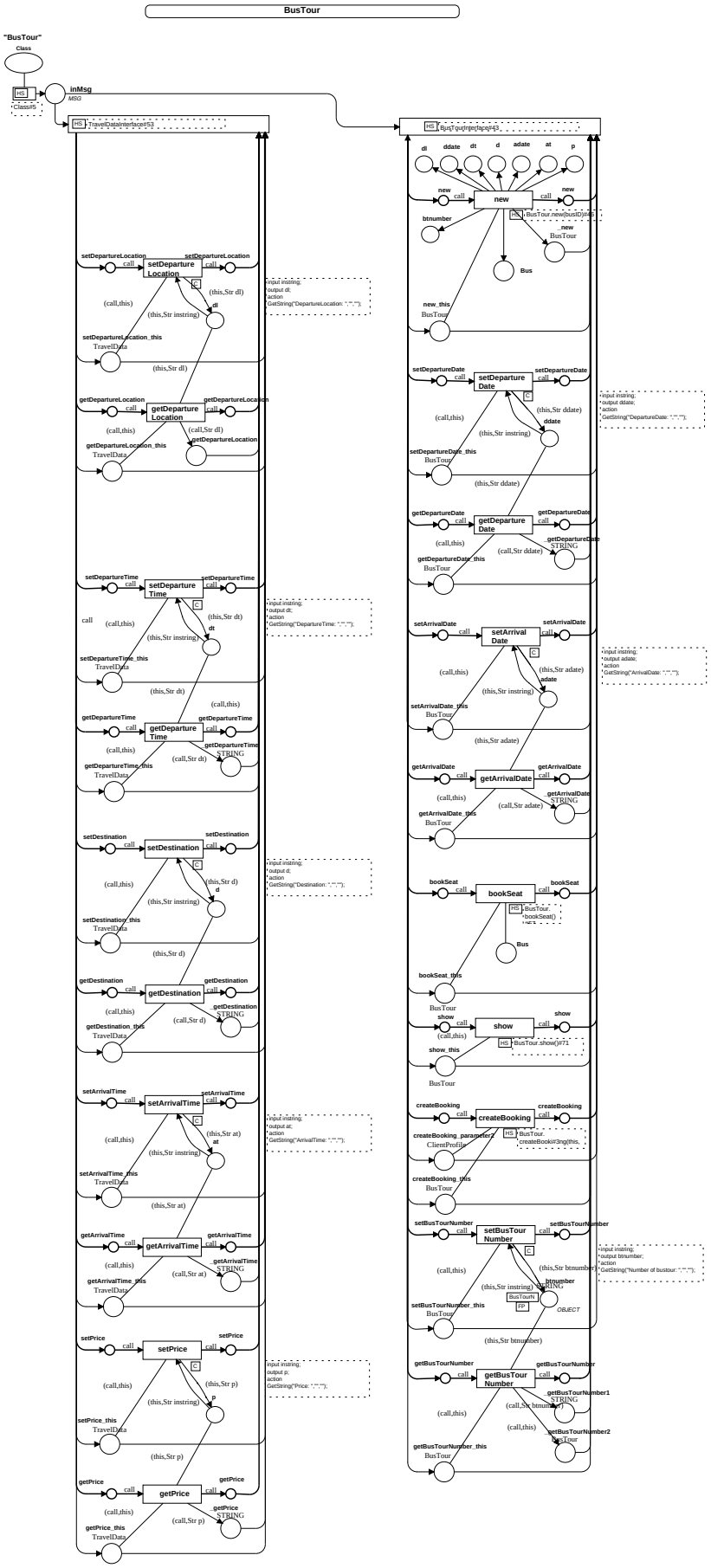

BusTourInterface
inMsg
MSG
ReturnMs
FG
MSG'
(call,[I(class,oid),M"new"])
[call,out1]
new
call
_new
BusTour
(call,out1)
new_this
BusTour
(call,I(class,oid))
(call,this)
(call,[I(class,oid),M"setDepartureDate"])
[call]
setDepartureDate
setDepartureDate_this
(call,[I(class,oid),M"getDepartureDate"])
[call,out1]
getDepartureDate
_getDepartureDate
STRING
getDepartureDate_this
(call,[I(class,oid),M"setArrivalDate"])
[call]
setArrivalDate
setArrivalDate_this
(call,[I(class,oid),M"getArrivalDate"])
[call,out1]
getArrivalDate
_getArrivalDate
getArrivalDate_this
(call,[I(class,oid),M"bookSeat"])
[call]
bookSeat
bookSeat_this
(call,[I(class,oid),M"show"])
[call]
show
show_this
(call,[I(class,oid),M"createBooking",bt,cl])
[call]
createBooking
createBooking_parameter1
(call,bt)
(call,bt)
createBooking_parameter2
Client
(call,cl)
(call,cl)
createBooking_this
(call,[I(class,oid),M"setBusTourNumber"])
[call]
setBusTourNumber
setBusTourNumber_this
(call,[I(class,oid),M"getBusTourNumber"])
[call,out1,out2]
getBusTourNumber
_getBusTourNumber1
_getBUSTourNumber2
(call,out2)
getBusTourNumber_this
P
In
Out
I/O


BusTour.show()
show_this
BusTour
P I/O
P In
show
call
fork
HS
dl
ddate
dt
d
adate
at
p
btnumber
"getDepartureLocation"
"getDepartureDate"
"getDepartureTime"
"getDestination"
"getArrivalDate"
"getArrivalTime"
"getPrice"
"getBusTourNumber"
this.getDepartureLocation()
this.getDepartureDate()
this.getDepartureTime()
this.getDestination()
this.getArrivalDate()
this.getArrivalTime()
this.getPrice()
this.getBusTourNumber()
join
call
(call,Str dl)
(call,Str ddate)
(call,Str dt)
(call,Str d)
(call,Str adate)
(call,Str at)
(call,Str p)
(call,Str btnumber)
C
input (dl,ddate,dt,d,adate,at,p,btnumber);
action
DSUI_UserAckMessage("BusTour No.
"^btnumber^": "^chr(10)^"Departure location:
"^dl^" - "^"Destination: "^d^""^chr(10)
^"Departure date: "^ddate^" - "^"Departure time:
"^dt^""^chr(10)^"Arrival date: "^adate^" -
"^"Arrival time: "^at^""^chr(10)^"Price: "^p^"");
P Out
show

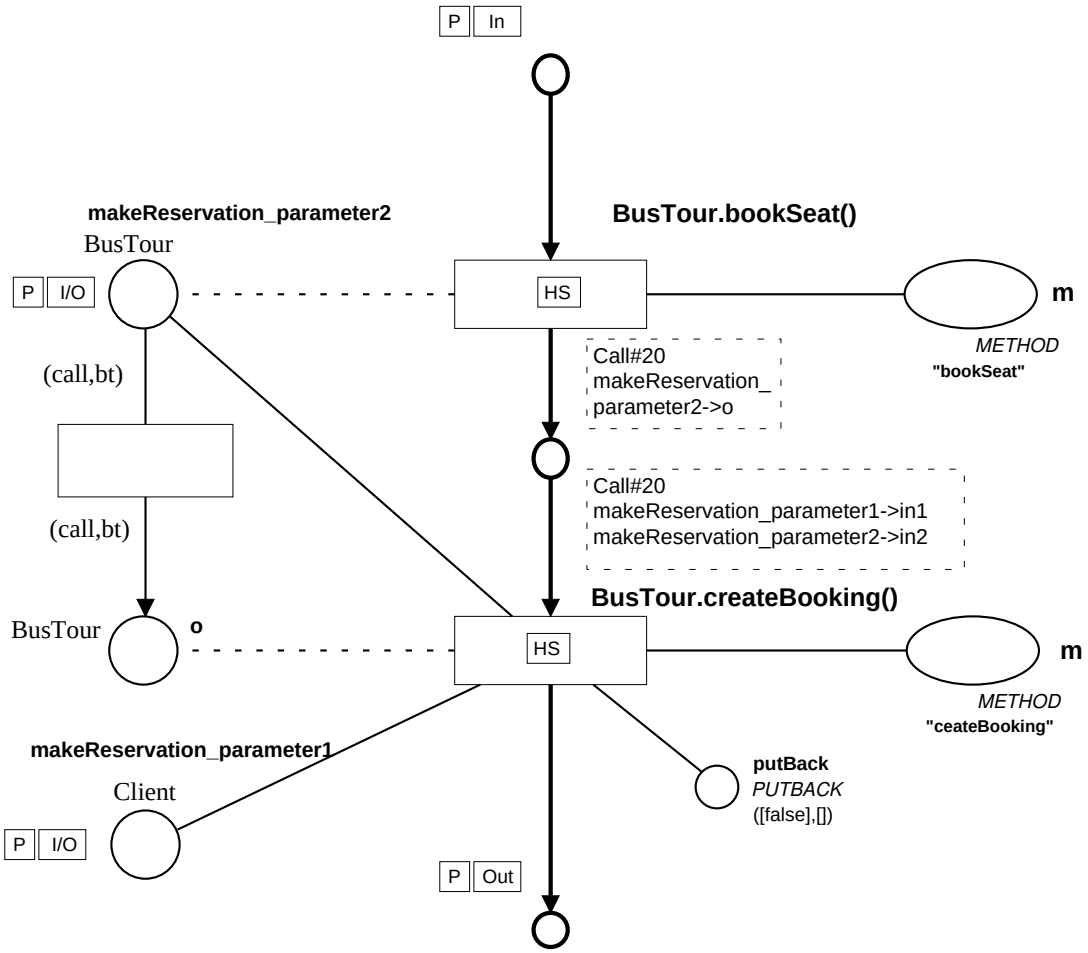

BusTour.bookSeat()
Bus
P
I/O
Bus
P
In
bookSeat
call
bookSeat_this
Bus.bookSeat()
m
P
I/O
HS
BusTour
"bookSeat"
call
Call#20
bookSeat->control
bookSeat->control
bookSeat_this->this
Bus->o
P
Out
bookSeat

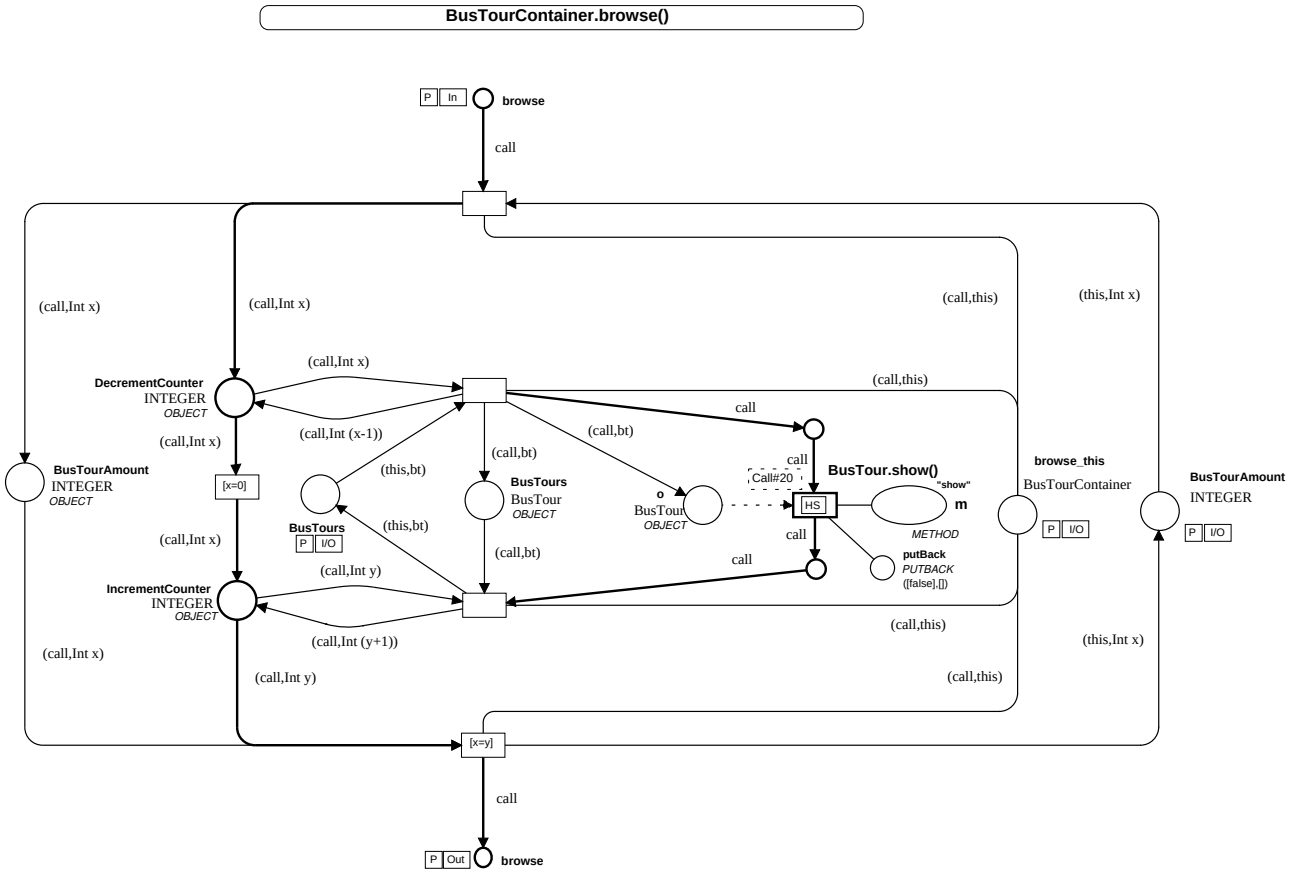

BusTourContainer.getBusTour(btno:Integer)
P In
control
OBJECT'
call
(this,Int x)
(call,this)
DecrementCounter
INTEGER
OBJECT
(this,Int (x-1))
(call,bt)
BusTourAmount
INTEGER
OBJECT
[x=0]
BusTours
BusTour
OBJECT
P I/O
(this,bt)
Call#20
HS
"getBusTourNumber"
m
METHOD
o
getBusTour_this
BusTourContainer
putBack
PUTBACK
([false],[])
IncrementCounter
(this,Int y)
(this,Int (y+1))
[x=y]
P Out
BusTour
bustournumber
(call,Str btnumber)
(call,Str btno1)
(call,Str btno2)
[btno1<>btno2]
[btno1=btno2]
UncorrectBusTour
CorrectBusTour


BusTour.new(busno:Integer)
P In
new
new_this
BusTour
P I/O
call
(call,this)
(this,Str "")
init
C
P Out
Bus
(this,Int busNo)
dl
ddate
dt
d
adate
at
p
btnumber
output busNo;
action
GetInteger("Please enter the number of
the bus for the new bustour: ",0,0);
call
fork
HS
"setDepartureLocation"
"setDepartureDate"
"setDepartureTime"
"setDestination"
"setArrivalDate"
"setArrivalTime"
"setPrice"
"setBusTourNumber"
this.setDepartureLocation()
this.setDepartureDate()
this.setDepartureTime()
this.setDestination()
this.setArrivalDate()
this.setArrivalTime()
this.setPrice()
this.setBusTourNumber()
join
call
(call,this)
(call,this)
_new
BusTour
call
new

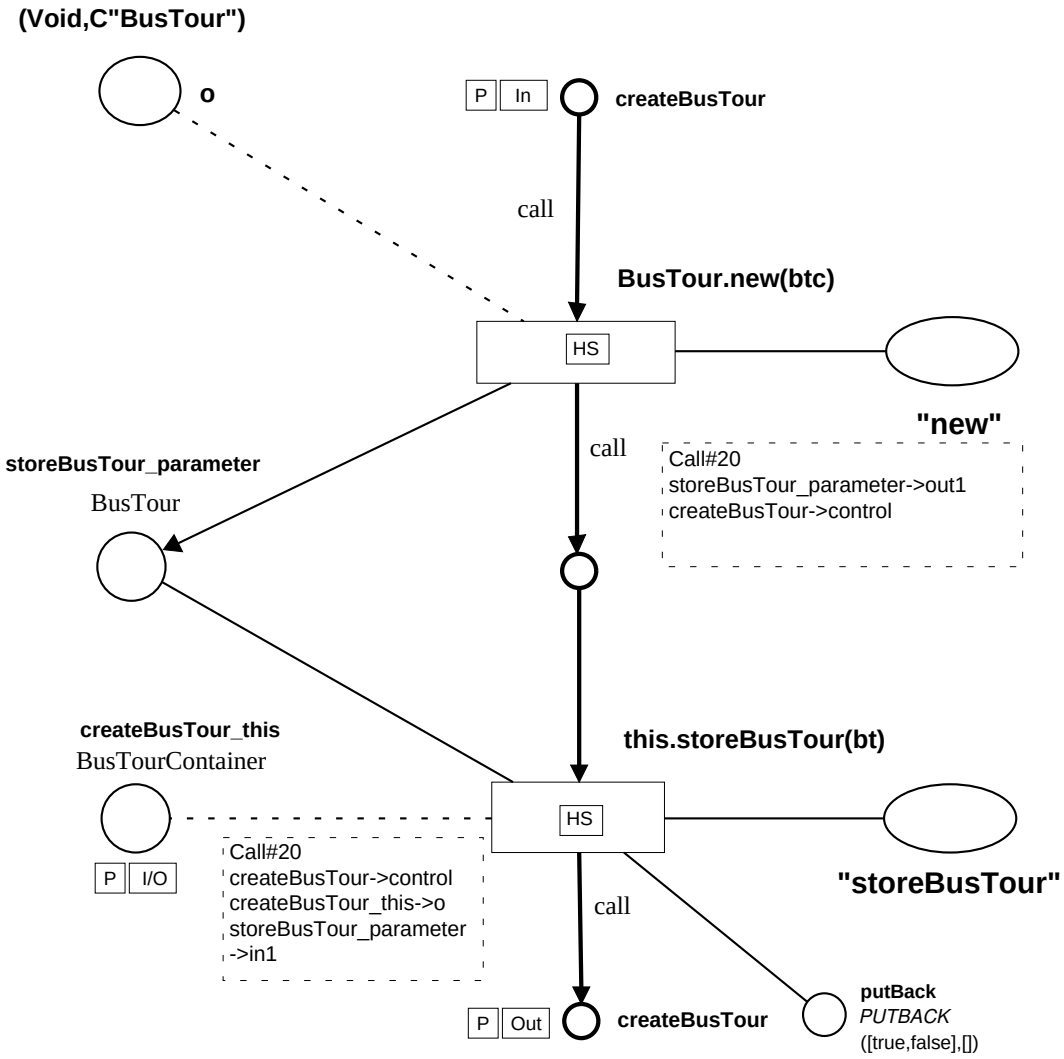

BusTour
"BusTour"
Class
HS
Class#5
inMsg
MSG
HS TravelDataInterface#53
HS BusTourInterface#43
setDepartureLocation
setDeparture Location
getDepartureLocation
getDeparture Location
setDepartureTime
setDeparture Time
getDepartureTime
getDeparture Time
setDestination
setDestination
getDestination
getDestination
setArrivalTime
setArrivalTime
getArrivalTime
getArrivalTime
setPrice
setPrice
getPrice
getPrice
new
BusTour.new(busID)#45
setDepartureDate
setDeparture Date
getDepartureDate
getDeparture Date
setArrivalDate
setArrival Date
getArrivalDate
getArrivalDate
bookSeat
bookSeat
show
BusTour.show()#71
createBooking
createBooking
setBusTourNumber
setBusTour Number
getBusTourNumber
getBusTour Number
BusTourN
FP
OBJECT
input instring;
output dl;
action
GetString("DepartureLocation: ","","");
input instring;
output dt;
action
GetString("DepartureTime: ","","");
input instring;
output d;
action
GetString("Destination: ","","");
input instring;
output at;
action
GetString("ArrivalTime: ","","");
input instring;
output p;
action
GetString("Price: ","","");
input instring;
output ddate;
action
GetString("DepartureDate: ","","");
input instring;
output adate;
action
GetString("ArrivalDate: ","","");
input instring;
output btnumber;
action
GetString("Number of bustour: ","","");

BusTourContainer.makeReservation()

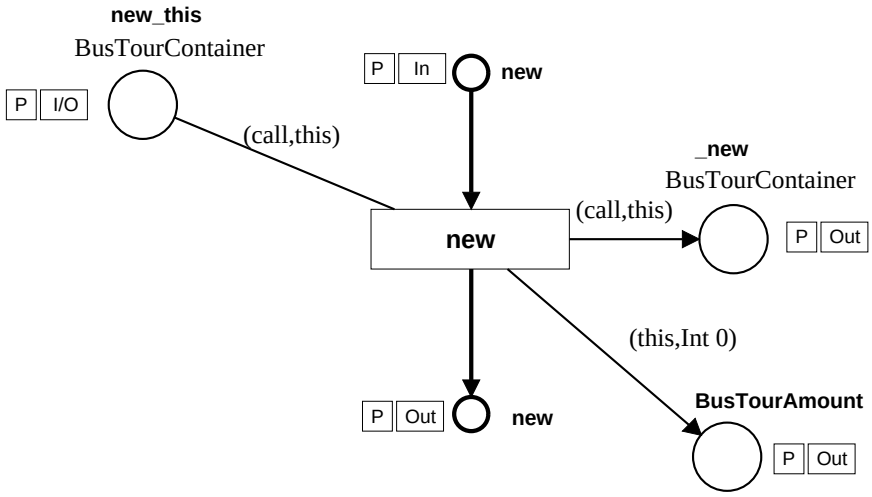


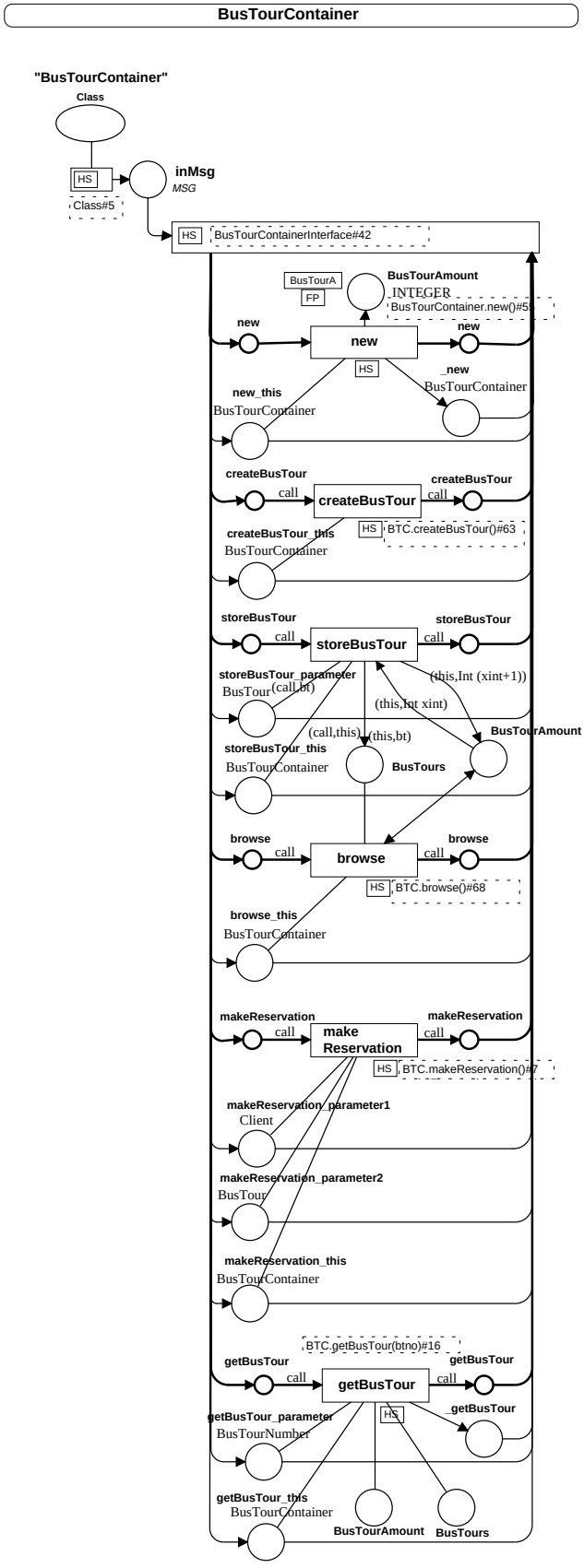

BusTourContainer.browse()
P
In
browse
call
(call,Int x)
(call,this)
(this,Int x)
DecrementCounter
INTEGER
OBJECT
(call,Int (x-1))
(call,bt)
(this,bt)
BusTourAmount
INTEGER
OBJECT
[x=0]
BusTours
BusTour
OBJECT
o
BusTour
OBJECT
Call#20
HS
BusTour.show()
"show"
m
METHOD
putBack
PUTBACK
([false].[])
browse_this
BusTourContainer
P
I/O
BusTourAmount
INTEGER
IncrementCounter
INTEGER
OBJECT
(call,Int y)
(call,Int (y+1))
[x=y]
Out
browse

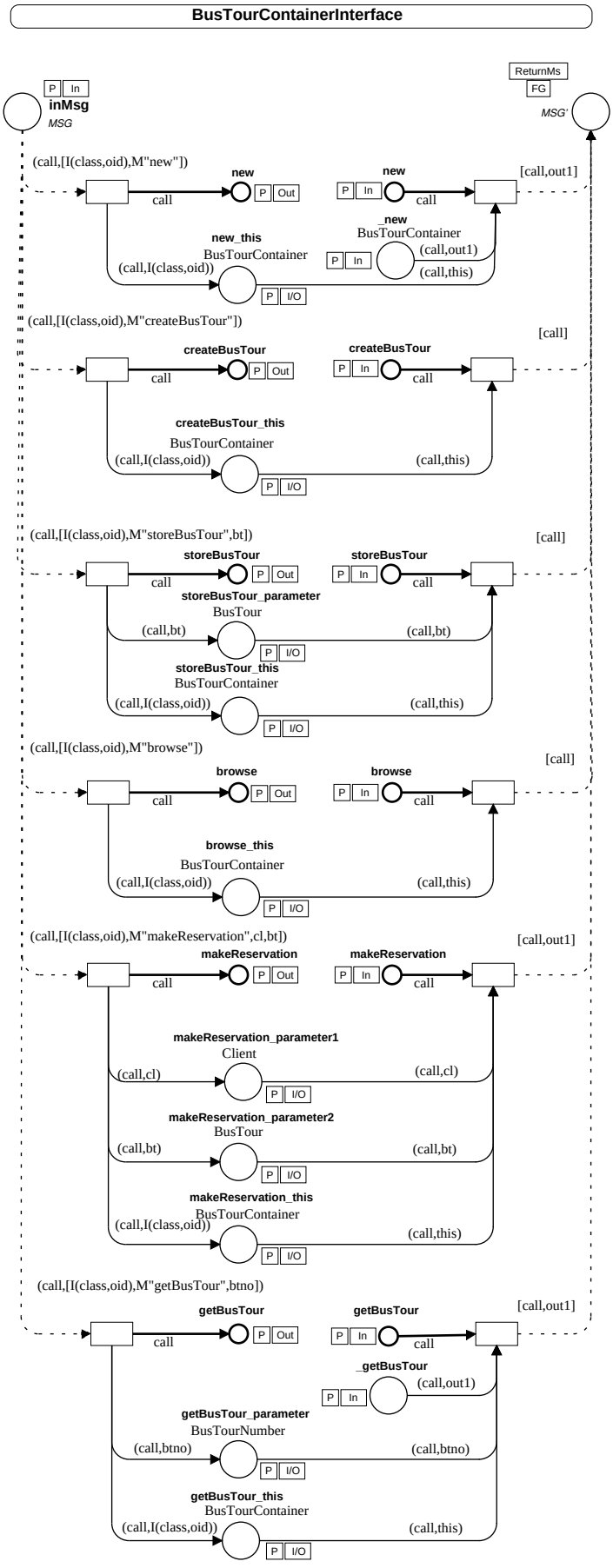
BusTourContainer.createBusTour()



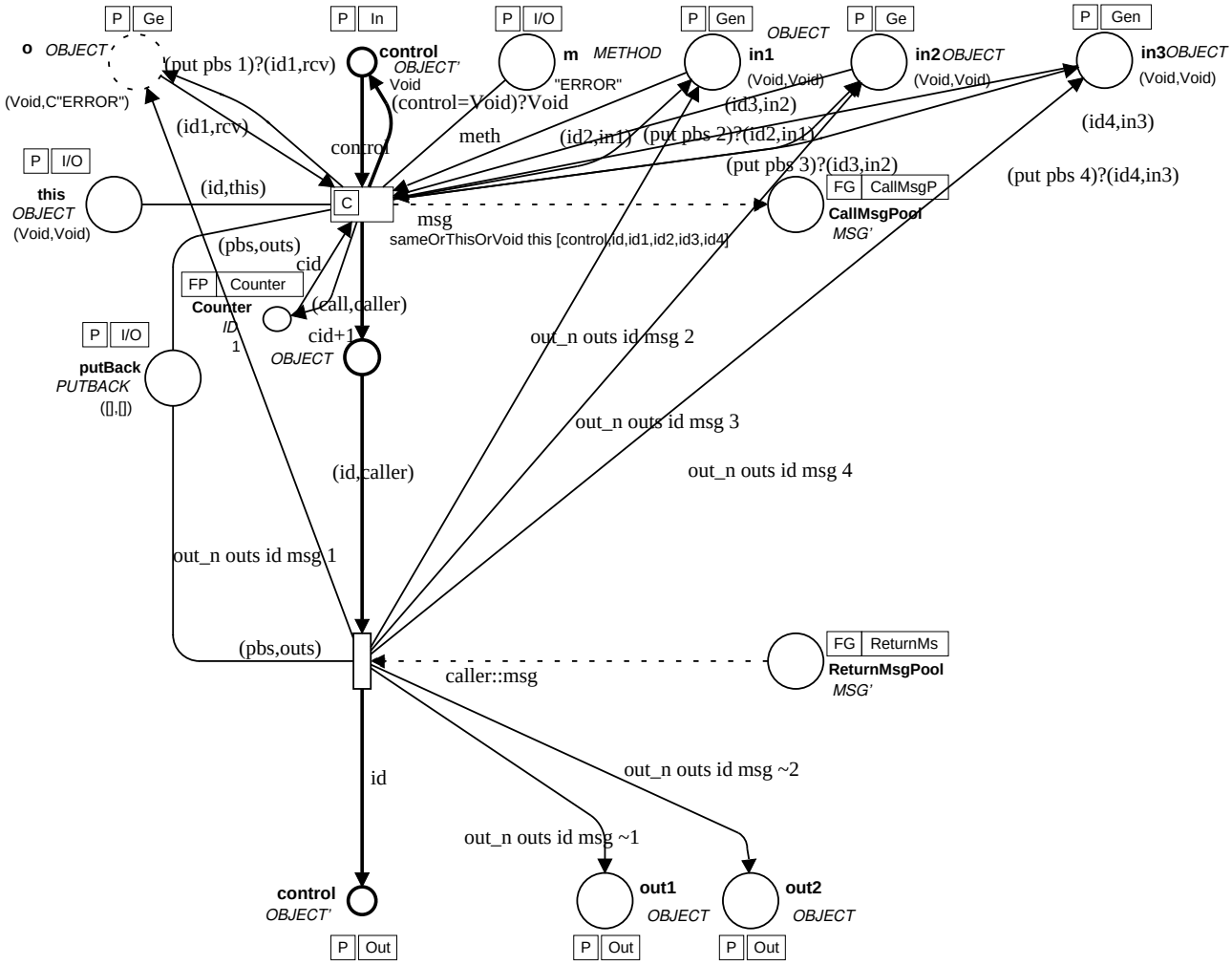
BusTourContainer.new()




BusTourContainer
"BusTourContainer"
Class
HS
Class#5
inMsg
MSG
HS
BusTourContainerInterface#42
BusTourA
FP
BusTourAmount
INTEGER
BusTourContainer.new()#5
new
new
new
HS
_new
BusTourContainer
new_this
BusTourContainer
createBusTour
call
createBusTour
call
createBusTour
HS
BTC.createBusTour()#63
createBusTour_this
BusTourContainer
storeBusTour
call
storeBusTour
call
storeBusTour
storeBusTour_parameter
BusTour
(call,bt)
(this,Int (xint+1))
(this,Int xint)
(call,this)
(this,bt)
BusTourAmount
storeBusTour_this
BusTourContainer
BusTours
browse
call
browse
call
browse
HS
BTC.browse()#68
browse_this
BusTourContainer
makeReservation
call
make
Reservation
call
makeReservation
HS
BTC.makeReservation()#7
makeReservation_parameter1
Client
makeReservation_parameter2
BusTour
makeReservation_this
BusTourContainer
BTC.getBusTour(btno)#16
getBusTour
call
getBusTour
call
getBusTour
HS
_getBusTour
getBusTour_parameter
BusTourNumber
getBusTour_this
BusTourContainer
BusTourAmount
BusTours

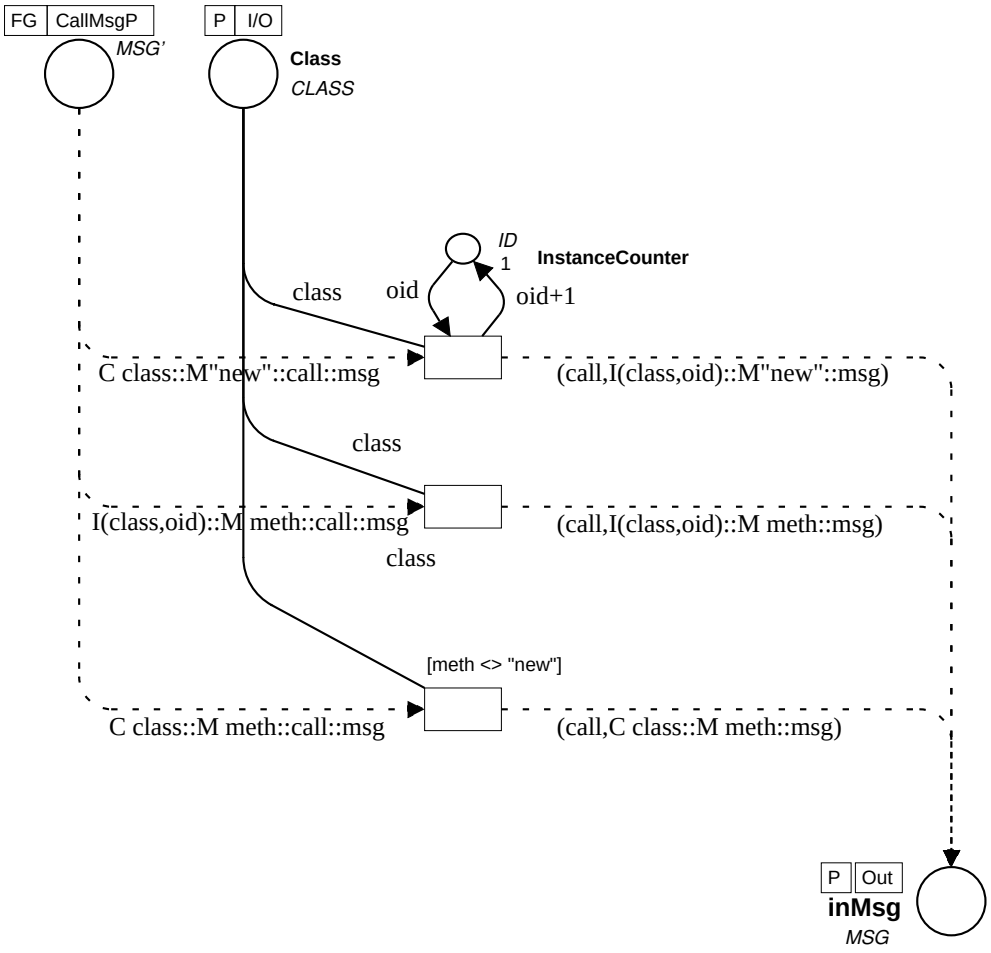

BusTourContainerInterface
P In
inMsg
MSG
ReturnMs
FG
MSG'
(call,[I(class,oid),M"new"])
new
call
P Out
new
P In
call
[call,out1]
_new
BusTourContainer
(call,out1)
P In
new_this
BusTourContainer
(call,I(class,oid))
(call,this)
P I/O
(call,[I(class,oid),M"createBusTour"])
[call]
createBusTour
call
P Out
createBusTour
P In
call
createBusTour_this
BusTourContainer
(call,I(class,oid))
(call,this)
P I/O
(call,[I(class,oid),M"storeBusTour",bt])
[call]
storeBusTour
call
P Out
storeBusTour
P In
call
storeBusTour_parameter
BusTour
(call,bt)
(call,bt)
P I/O
storeBusTour_this
BusTourContainer
(call,I(class,oid))
(call,this)
P I/O
(call,[I(class,oid),M"browse"])
[call]
browse
call
P Out
browse
P In
call
browse_this
BusTourContainer
(call,I(class,oid))
(call,this)
P I/O
(call,[I(class,oid),M"makeReservation",cl,bt])
[call,out1]
makeReservation
call
P Out
makeReservation
P In
call
makeReservation_parameter1
Client
(call,cl)
(call,cl)
P I/O
makeReservation_parameter2
BusTour
(call,bt)
(call,bt)
P I/O
makeReservation_this
BusTourContainer
(call,I(class,oid))
(call,this)
P I/O
(call,[I(class,oid),M"getBusTour",btno])
[call,out1]
getBusTour
call
P Out
getBusTour
P In
call
_getBusTour
(call,out1)
P In
getBusTour_parameter
BusTourNumber
(call,btno)
(call,btno)
P I/O
getBusTour_this
BusTourContainer
(call,I(class,oid))
(call,this)
P I/O

Call-Page




```
input (control,id,this,id1,rcv,id2,meth,id3,in1,in2,cid,in3,id4);
output(call,caller,msg);
action
  let val cntrl = nonThisOrVoid this [control,id,id1,id2,id3,id4];
    val caller = I("Call",cid);
in (cntrl,caller,rcv::M meth::caller::(noVoid [in1,in2,in3]))
end;
```

Class-Page


FG
CallMsgP
MSG'
P
I/O
Class
CLASS
ID
1
InstanceCounter
class
oid
oid+1
C class::M"new"::call::msg
(call,I(class,oid)::M"new"::msg)
class
I(class,oid)::M meth::call::msg
(call,I(class,oid)::M meth::msg)
class
[meth <> "new"]
C class::M meth::call::msg
(call,C class::M meth::msg)
P
Out
inMsg
MSG

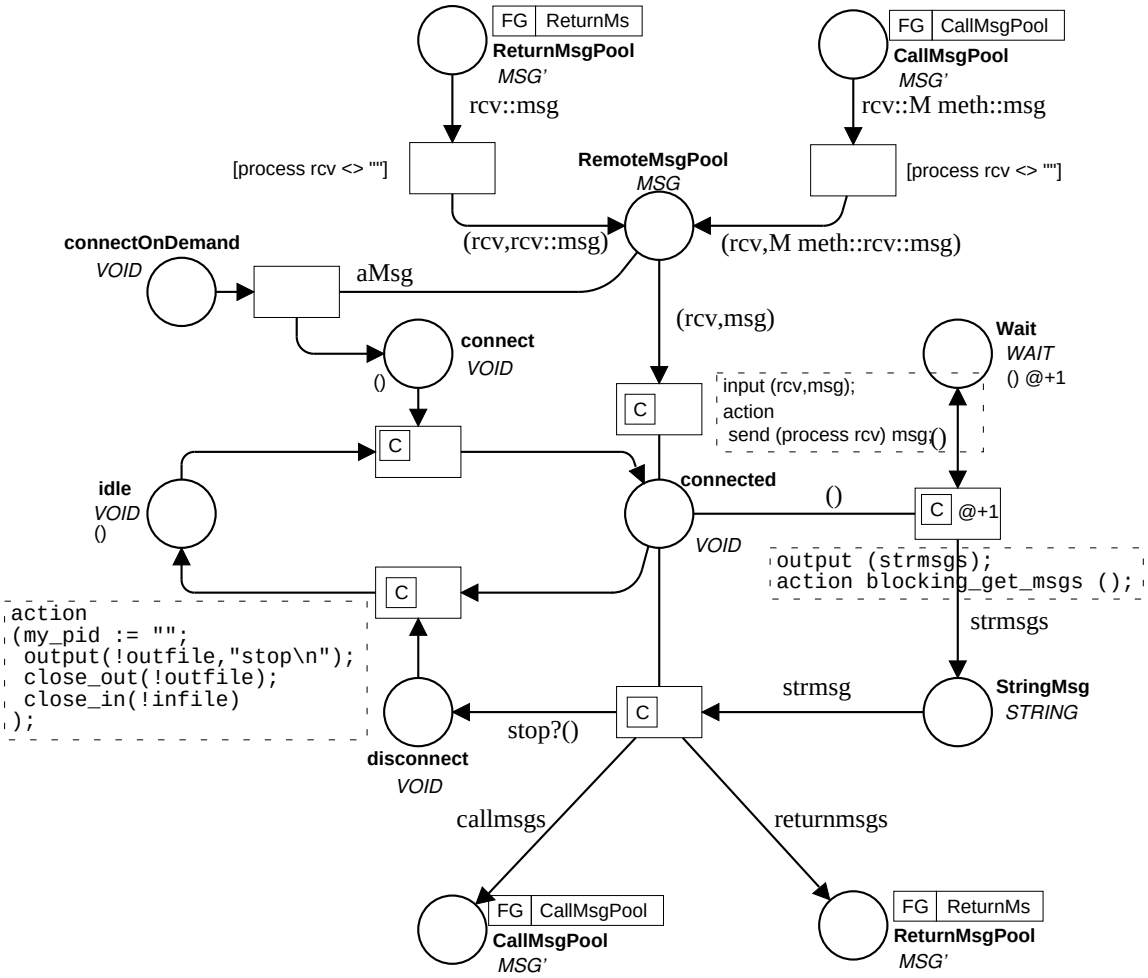
Code-Page

```
(* workaround: *)
val my_execute = execute;

color WAIT = unit timed;
color VOID = unit;
color INTEGER = int;
color REAL = real;
color STRING = string;
color BOOLEAN = bool;
color ID = INTEGER declare input_col;
color RID = STRING;
color BOOLEANLIST = list BOOLEAN;
color INTEGERLIST = list INTEGER;
color PUTBACK = product BOOLEANLIST * INTEGERLIST;
color VALUE = with Integer | RealNumber | String | Boolean;
color MODIFIER = with Static;
color MODIFIERS = list MODIFIER;
color CLASS = STRING;
color METHOD = STRING;
color INSTANCE = product CLASS * ID;
color RINSTANCE = product CLASS * RID * RID;
color RCLASS = product CLASS * RID;
color OBJECT' = union Void + Null
    + C:CLASS + RC:RCLASS
    + M:METHOD
    + I:INSTANCE + RI:RINSTANCE
    + Int:INTEGER + Str:STRING + Bool:BOOLEAN + Real:REAL
    declare mkst_col, of_M;
color MSG' = list OBJECT' declare input_col;
color TYPE = union Object + Reference + ClassRef + InstRef + Value
    + Method + AnyRef + Bottom
    + TC:CLASS + TI:CLASS + TM:METHOD + TV:VALUE;
color TYPELIST = list TYPE;
color MSGTYPE = product TYPELIST * TYPELIST;
color MSGDEF = product MODIFIERS * TYPELIST * METHOD * TYPELIST;
color RETURN' = product OBJECT' * TYPELIST;
color RETURN = product OBJECT'  * RETURN';
color MSG = product OBJECT'  * MSG';
color OBJECTLIST = MSG';
color OBJECT = product OBJECT'  * OBJECT';
color RIDxMSG' = product RID * MSG';

fun GetString(cr_prompt:string,cr_def:string,canc:string)=
let val Stringvalue =
DSUI_GetString{prompt=cr_prompt,def=cr_def}
in
  if Stringvalue <> "" then
      Stringvalue
  else
    GetString(cr_prompt,cr_def,canc)
  end
  handle _ => canc;

fun GetInteger(cr_prompt:string,cr_def:int,canc:int)=
let val Integervalue =
DSUI_GetIntegerValue{prompt=cr_prompt,def=cr_def}
in
  if Integervalue > 0 then
      Integervalue
  else
    GetInteger(cr_prompt,cr_def,canc)
  end
  handle _ => canc;



(* Net Inscription Abbreviations: *)
infix 5 ?;
fun p?v = if p then 1'v else empty;

infix 8 // ;
fun x // y = (x,y);

local fun mkst_list' _ nil = ""
   | mkst_list' mkst [elem] = mkst elem
   | mkst_list' mkst (elem::lst) = mkst elem ^ "," ^ mkst_list' mkst lst
in fun mkst_list mkst lst = "[" ^ mkst_list' mkst lst ^ "]" end;

val msgToStr = mkst_list mkst_col'OBJECT';

fun strToMsg strmsg = (input_col'MSG' (open_string strmsg)) handle _ => [];

local fun remLocal' pid (C(class)) = RC(class,pid)
   | remLocal' pid (I(class,oid)) = RI(class,pid,makestring oid)
   | remLocal' _ old = old
in fun remLocal pid msg = map (remLocal' pid) msg end;

local fun toLocal' pid (old as RC(class,pid')) = if pid=pid' then C(class) else old
   | toLocal' pid (old as RI(class,pid',roid)) =
     if pid=pid' then I(class,input_col'ID (open_string roid)) else old
   | toLocal' _ old = old
in fun toLocal pid msg = map (toLocal' pid) msg end;


fun getType Void = Bottom
   | getType Null = AnyRef
   | getType (C class) = TC class
   | getType (RC(class,_)) = TC class
   | getType (M meth) = TM meth
   | getType (I(class,_)) = TI class
   | getType (RI(class,_,_)) = TI class
   | getType (Int _) = TV Integer
   | getType (Str _) = TV String
   | getType (Bool _) = TV Boolean
   | getType (Real _) = TV RealNumber;

infix 8 <== ;
fun type1  <== type2 =
 if type1 = type2 then true
 else case (type1,type2) of
   (_,        Object   ) => true
   | (Bottom,          ) => true
   | (AnyRef, TC _     ) => true
   | (AnyRef, TI _     ) => true
   | (AnyRef, ClassRef ) => true
   | (AnyRef, InstRef  ) => true
   | (AnyRef, Reference) => true
   | (TC _  , ClassRef ) => true
   | (TC _  , Reference) => true
   | (TI _  , InstRef  ) => true
   | (TI _  , Reference) => true
   | (TM _  , Method   ) => true
   | (TV _  , Value    ) => true
   | (_     , _        ) => false;

fun ofType typ obj = getType obj <== typ;
val ofUser = ofType (TI "User");
val ofTail = ofType (TI "tail");
val ofPair = ofType (TI "Pair");


fun typeCheck nil nil = true
  | typeCheck (typ::types) (obj::objs) =
 ofType typ obj andalso typeCheck types objs
  | typeCheck _ _ = false;

fun nonThisOrVoid _ nil = Void (* darf nicht vorkommen! *)
  | nonThisOrVoid this (x::objs) = if x=Void orelse x=this then nonThisOrVoid this objs
          else x;

local
fun sameOrThisOrVoid' y _ nil = y <> Void
  | sameOrThisOrVoid' Void this (x::objs) =
 if x=Void orelse x=this then sameOrThisOrVoid' Void this objs
         else sameOrThisOrVoid' x this objs
  | sameOrThisOrVoid' y this (x::objs) =
 if x=Void orelse x=this orelse x=y then sameOrThisOrVoid' y this objs
         else false
in val sameOrThisOrVoid = sameOrThisOrVoid' Void end;

fun noVoid nil = nil
  | noVoid (Void::xs) = noVoid xs
  | noVoid (x::xs) = x::noVoid xs;

fun prefix xs 0 = nil
  | prefix (x::xs) n = x::prefix xs (n-1)
  | prefix nil _ = nil;

fun put (x::xs) 1 = x
  | put (x::xs) n = put xs (n-1)
  | put nil _ = true;

fun in_n call (obj::objs) 1 = 1'(call,obj)
  | in_n call (_::objs) n = in_n call objs (n-1)
  | in_n call nil _ = empty; (* 1'(call,Null); *)

local fun position' _  nil _ = 0
  | position' n (x::xs) y = if x = y then n else position' (n+1) xs y
in val position = position' 1
end;

fun out_n outs call msg index =
 if mem outs index
  orelse (length outs < -index andalso length msg >= -index)
 then let val pos = if mem outs index then position outs index
       else -index
   in 1'(call,nth(msg,pos-1)) end
 else empty;

fun process (RC(_,pid)) = pid
  | process (RI(_,pid,_)) = pid
  | process _ = 0;

globref my_pid = "";
globref infile = std_in;
globref outfile = std_out;

fun send pid msg =
 (output(!outfile, "send\n"^ pid ^"\n" ^
  msgToStr(remLocal (!my_pid) msg) ^"\n");
    flush_out(!outfile));

val callcnt = ref 0;

infix 8 \;
fun (obj\meth) params =
  let val caller = I("CALL",(callcnt:= (!callcnt)+1;!callcnt))
  in (send (process (hd (remLocal (!my_pid) [obj])))
           (M meth::obj::caller::params);
            caller)
  end;

infix 8 return;
fun (obj) return params) =
  (send (process (hd (remLocal (!my_pid) [obj])))
           (obj::params));

fun blocking_get_msgs () =
     (1,input_line(!infile)) !!
   (if can_input(!infile)>0
  then blocking_get_msgs ()
  else empty);
```

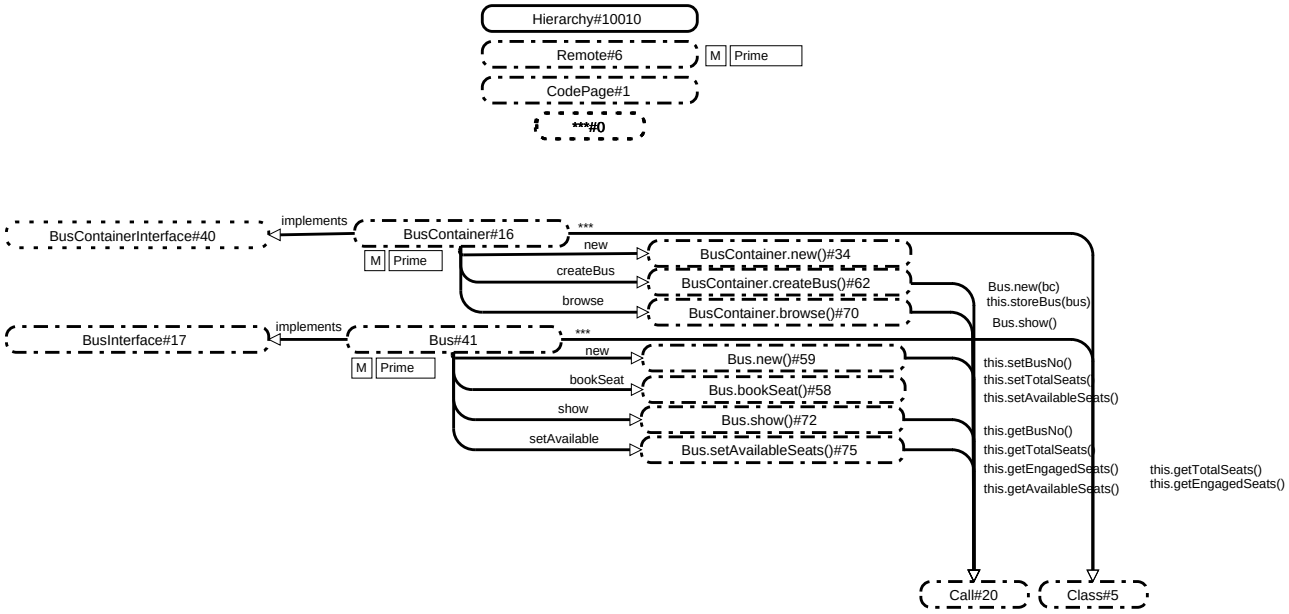
```
var localmsgs, callmsgs, returnmsgs: MSG' ms;
var strmsg: STRING;
var strmsgs: STRING ms;
var timer, timer': INTEGER;
var pid,roid: RID;
var pbs: BOOLEANLIST;
var outs: INTEGERLIST;
var msg: MSG';
var aMsg: MSG;
var onDemand, stop: BOOLEAN;
var class: CLASS;
var msgType: MSGTYPE;
var inClasses, outClasses: TYPELIST;
var cid,oid: ID;
var control,call,caller,
    id, id', id1, id2, id3, id4: OBJECT';
var meth: METHOD;
var obj, in1,in2,in3,out1,out2: OBJECT';
var this,classobj,snd,rcv,
 lst,pair,cons,tail,add,
    filter,filteredList,
 mapObj,filterObj: OBJECT';
var newHead: OBJECT';
var head,car,cdr,loginstring, parameter, parameter1, parameter2: OBJECT';
var isEmpty, isString, toString, result:OBJECT';
var this'cons,this'cons',
 this'left,this'left',
 this'right,this'right': OBJECT';

var tsc,btc,bc,bus,ts,bt,btno,busID,cl:OBJECT';
var instring,choicestring,dday,aday,dl,dt,d,at,p,btno1,btno2,btnumber,
adate,ddate:STRING;
var x,y,inint,xint,yint,zint,tSeats,aSeats,eSeats,busNo:INTEGER;
var object, choice, loginout:OBJECT;
```

Remote-Page

```
action
  let val name = "BusTourProcess";
   val (inf,outf) = my_execute("/home/tgi_3/tgi3/bin/messenger",[name]);
  in (my_pid := name; infile := inf; outfile := outf) end;
```




```
input (strmsg);
output (callmsgs,returnmsgs,stop);
action
 case toLocal (!my_pid) (strToMsg (strmsg)) of
    M(m)::rcv::param => (1‘(rcv::M(m)::param), empty, false)
  | I(i)::msg        => (empty,          1‘(I(i)::msg), false)
  | msg              => (empty,          1‘msg,         true);
```


Hierarchy#10010
Remote#6
M
Prime
CodePage#1
***#0
BusContainerInterface#40
implements
BusContainer#16
M
Prime
***
new
createBus
browse
BusContainer.new()#34
BusContainer.createBus()#62
BusContainer.browse()#70
Bus.new(bc)
this.storeBus(bus)
Bus.show()
BusInterface#17
implements
Bus#41
M
Prime
***
new
bookSeat
show
setAvailable
Bus.new()#59
Bus.bookSeat()#58
Bus.show()#72
Bus.setAvailableSeats()#75
this.setBusNo()
this.setTotalSeats()
this.setAvailableSeats()
this.getBusNo()
this.getTotalSeats()
this.getEngagedSeats()
this.getAvailableSeats()
this.getTotalSeats()
this.getEngagedSeats()
Call#20
Class#5

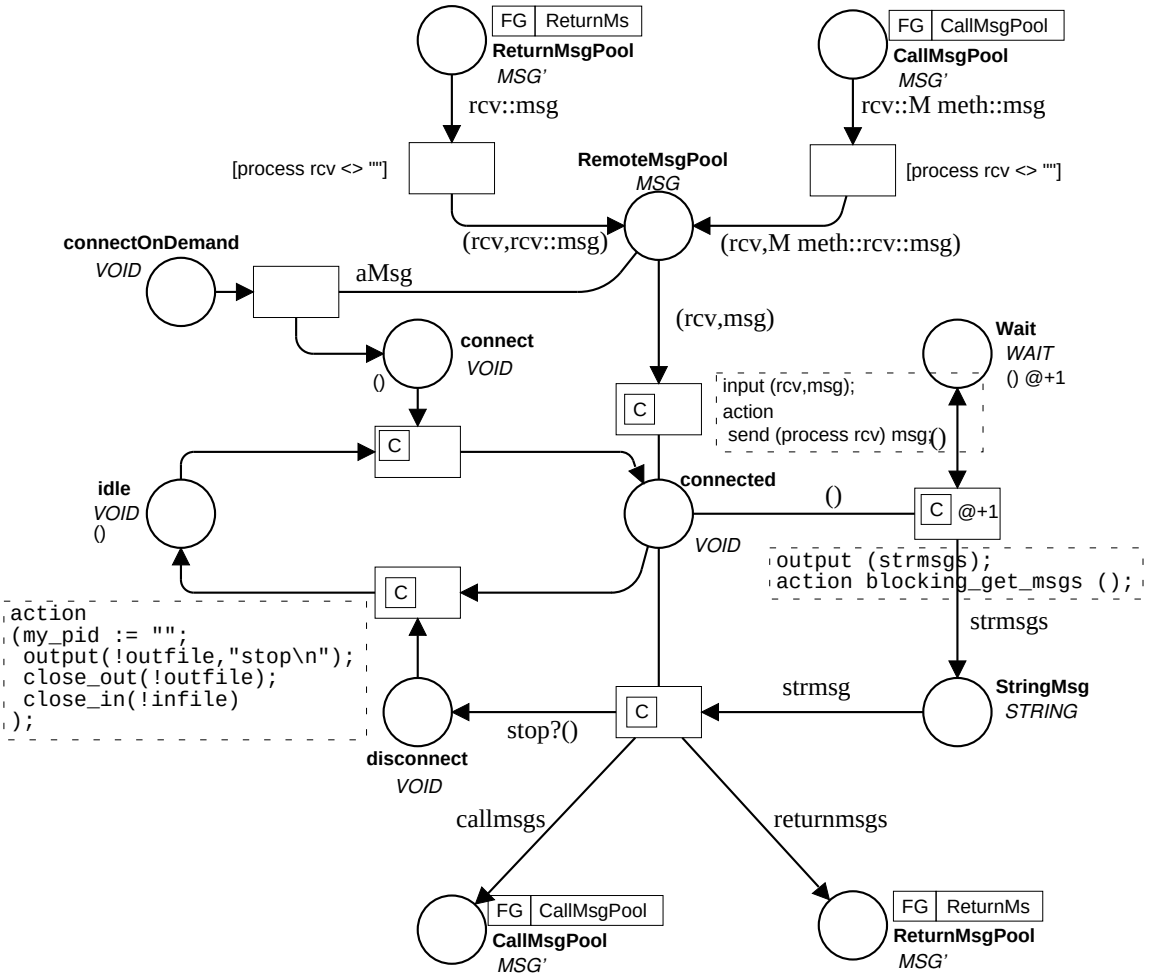
Code-Page

```
(* workaround: *)
val my_execute = execute;

color WAIT = unit timed;
color VOID = unit;
color INTEGER = int;
color REAL = real;
color STRING = string;
color BOOLEAN = bool;
color ID = INTEGER declare input_col;
color RID = STRING;
color BOOLEANLIST = list BOOLEAN;
color INTEGERLIST = list INTEGER;
color PUTBACK = product BOOLEANLIST * INTEGERLIST;
color VALUE = with Integer | RealNumber | String | Boolean;
color MODIFIER = with Static;
color MODIFIERS = list MODIFIER;
color CLASS = STRING;
color METHOD = STRING;
color INSTANCE = product CLASS * ID;
color RINSTANCE = product CLASS * RID * RID;
color RCLASS = product CLASS * RID;
color OBJECT = union Void + Null
    + C:CLASS + RC:RCLASS
    + M:METHOD
    + I:INSTANCE + RI:RINSTANCE
    + Int:INTEGER + Str:STRING + Bool:BOOLEAN + Real:REAL
    declare mkst_col, of_M;
color MSG' = list OBJECT' declare input_col;
color TYPE = union Object + Reference + ClassRef + InstRef + Value
    + Method + AnyRef + Bottom
    + TC:CLASS + TI:CLASS + TM:METHOD + TV:VALUE;
color TYPELIST = list TYPE;
color MSGTYPE = product TYPELIST * TYPELIST;
color MSGDEF = product MODIFIERS * TYPELIST * METHOD * TYPELIST;
color RETURN' = product OBJECT' * TYPELIST;
color RETURN = product OBJECT'  * RETURN';
color MSG = product OBJECT'  * MSG';
color OBJECTLIST = MSG;
color OBJECT = product OBJECT'  * OBJECT';
color RIDxMSG' = product RID * MSG';

fun GetString(cr_prompt:string,cr_def:string,canc:string)=
let val Stringvalue =
DSUI_GetString{prompt=cr_prompt,def=cr_def}
in
  if Stringvalue <> "" then
        Stringvalue
  else
    GetString(cr_prompt,cr_def,canc)
  end
  handle _ => canc;

fun GetInteger(cr_prompt:string,cr_def:int,canc:int)=
let val Integervalue =
DSUI_GetIntegervalue{prompt=cr_prompt,def=cr_def}
in
  if Integervalue > 0 then
        Integervalue
  else
    GetInteger(cr_prompt,cr_def,canc)
  end
  handle _ => canc;



(* Net Inscription Abbreviations: *)
infix 5 ?;
fun p?v = if p then 1'v else empty;

infix 8 // ;
fun x // y = (x,y);

local fun mkst_list' _ nil = ""
  | mkst_list' mkst [elem] = mkst elem
  | mkst_list' mkst (elem::lst) = mkst elem ^ "," ^ mkst_list' mkst lst
in fun mkst_list mkst lst = "[" ^ mkst_list' mkst lst ^ "]" end;

val msgToStr = mkst_list mkst_col'OBJECT';

fun strToMsg strmsg = (input_col'MSG' (open_string strmsg)) handle _ => [];

local fun remLocal' pid (C(class)) = RC(class,pid)
  | remLocal' pid (I(class,oid)) = RI(class,pid,makestring oid)
  | remLocal' _ old = old
in fun remLocal pid msg = map (remLocal' pid) msg end;

local fun toLocal' pid (old as RC(class,pid')) = if pid=pid' then C(class) else old
  | toLocal' pid (old as RI(class,pid',roid)) =
    if pid=pid' then I(class,input_col'ID (open_string roid)) else old
  | toLocal' _ old = old
in fun toLocal pid msg = map (toLocal' pid) msg end;


fun getType Void = Bottom
  | getType Null = AnyRef
  | getType (C class) = TC class
  | getType (RC(class,_)) = TC class
  | getType (M meth) = TM meth
  | getType (I(class,_)) = TI class
  | getType (RI(class,_,_)) = TI class
  | getType (Int _) = TV Integer
  | getType (Str _) = TV String
  | getType (Bool _) = TV Boolean
  | getType (Real _) = TV RealNumber;

infix 8 <== ;
fun type1  <== type2 =
 if type1 = type2 then true
 else case (type1,type2) of
   (_,          Object  ) => true
    | (Bottom,          ) => true
    | (AnyRef, TC _     ) => true
    | (AnyRef, TI _     ) => true
    | (AnyRef, ClassRef ) => true
    | (AnyRef, InstRef  ) => true
    | (AnyRef, Reference) => true
    | (TC _  , ClassRef ) => true
    | (TC _  , Reference) => true
    | (TI _  , InstRef  ) => true
    | (TI _  , Reference) => true
    | (TM _  , Method   ) => true
    | (TV _  , Value    ) => true
    | (_     ,          ) => false;

fun ofType typ obj = getType obj <== typ;
val ofUser = ofType (TI "User");
val ofTail = ofType (TI "Tail");
val ofPair = ofType (TI "Pair");


fun typeCheck nil nil = true
  | typeCheck (typ::types) (obj::objs) =
 ofType typ obj andalso typeCheck types objs
  | typeCheck _ _ = false;

fun nonThisOrVoid _ nil = Void (* darf nicht vorkommen! *)
  | nonThisOrVoid this (x::objs) = if x=Void orelse x=this then nonThisOrVoid this objs
          else x;

local
fun sameOrThisOrVoid' y _ nil = y <> Void
  | sameOrThisOrVoid' Void this (x::objs) =
  if x=Void orelse x=this then sameOrThisOrVoid' Void this objs
        else sameOrThisOrVoid' x this objs
  | sameOrThisOrVoid' y this (x::objs) =
  if x=Void orelse x=this orelse x=y then sameOrThisOrVoid' y this objs
        else false
in val sameOrThisOrVoid = sameOrThisOrVoid' Void end;

fun noVoid nil = nil
  | noVoid (Void::xs) = noVoid xs
  | noVoid (x::xs) = x::noVoid xs;

fun prefix xs 0 = nil
  | prefix (x::xs) n = x::prefix xs (n-1)
  | prefix nil _ = nil;

fun put (x::xs) 1 = x
  | put (x::xs) n = put xs (n-1)
  | put nil _ = true;

fun in_n call (obj::objs) 1 = 1'(call,obj)
  | in_n call (_::objs) n = in_n call objs (n-1)
  | in_n call nil _ = empty; (* 1'(call,Null); *)

local fun position' _ nil _ = 0
  | position' n (x::xs) y = if x = y then n else position' (n+1) xs y
in val position = position' 1
end;

fun out_n outs call msg index =
 if mem outs index
  orelse (length outs < ~index andalso length msg >= ~index)
 then let val pos = if mem outs index then position outs index
      else ~index
   in 1'(call,nth(msg,pos-1)) end
 else empty;

fun process (RC(_,pid)) = pid
  | process (RI(_,pid,_)) = pid
  | process _ = "";

globref my_pid = "";
globref infile = std_in;
globref outfile = std_out;

fun send pid msg =
 (output(!outfile, "send\n"^ pid ^"\n" ^
  msgToStr(remLocal (!my_pid) msg) ^"\n");
    flush_out(!outfile));

val callcnt = ref 0;

infix 8 \;
fun (obj\meth) params =
  let val caller = I("CALL",(callcnt:= (!callcnt)+1;!callcnt))
  in (send (process (hd (remLocal (!my_pid) [obj])))
           (M meth::obj::caller::params);
           caller)
  end;

infix 8 return;
fun (obj return params) =
  (send (process (hd (remLocal (!my_pid) [obj])))
        (obj::params));

fun blocking_get_msgs () =
    (1,input_line(!infile)) !!
    (if can_input(!infile)>0
  then blocking_get_msgs ()
  else empty);
```

```
var localmsgs, callmsgs, returnmsgs: MSG' ms;
var strmsg: STRING;
var strmsgs: STRING ms;
var timer, timer': INTEGER;
var pid,roid: RID;
var pbs: BOOLEANLIST;
var outs: INTEGERLIST;
var msg: MSG';
var aMsg: MSG;
var onDemand, stop: BOOLEAN;
var class: CLASS;
var msgType: MSGTYPE;
var inClasses, outClasses: TYPELIST;
var cid,oid: ID;
var control,call,caller,
    id, id', id1, id2, id3, id4: OBJECT';
var meth: METHOD;
var obj, in1,in2,in3,out1,out2: OBJECT';
var this,classobj,snd,rcv,
 lst,pair,cons,tail,add,
    filter,filteredlist,
 mapObj,filterObj: OBJECT';
var newHead: OBJECT';
var head,car,cdr,loginstring, parameter, parameter1, parameter2: OBJECT';
var isEmpty, isString, toString, result:OBJECT';
var this'cons,this'cons',
 this'left,this'left',
 this'right,this'right': OBJECT';

var x,y,tsc,btc,bc,bus,ts,bt,busID,cl:OBJECT';
var instring,choicestring,dday,aday,d1,dt,d,at,p,adate,ddate:STRING;
var inint,xint,yint,zint,tSeats,aSeats,eSeats,busNo:INTEGER;
var object, choice, loginout:OBJECT;
```

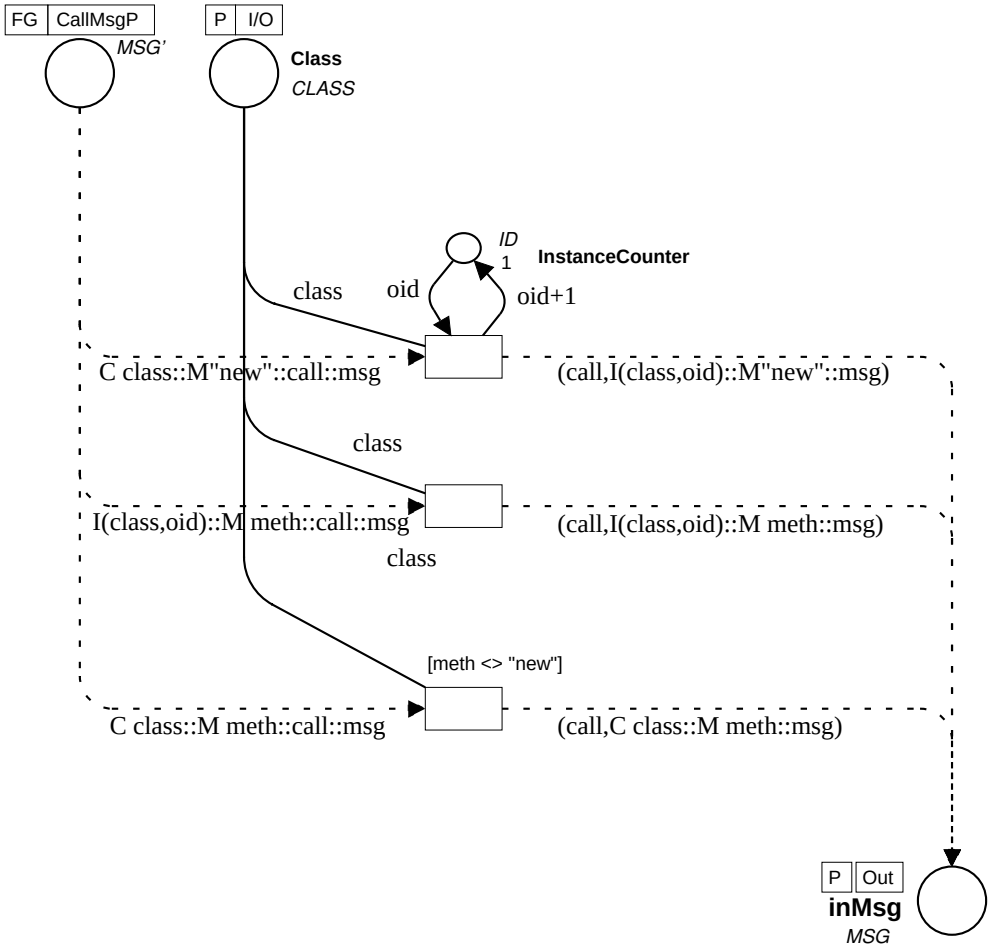
Remote-Page

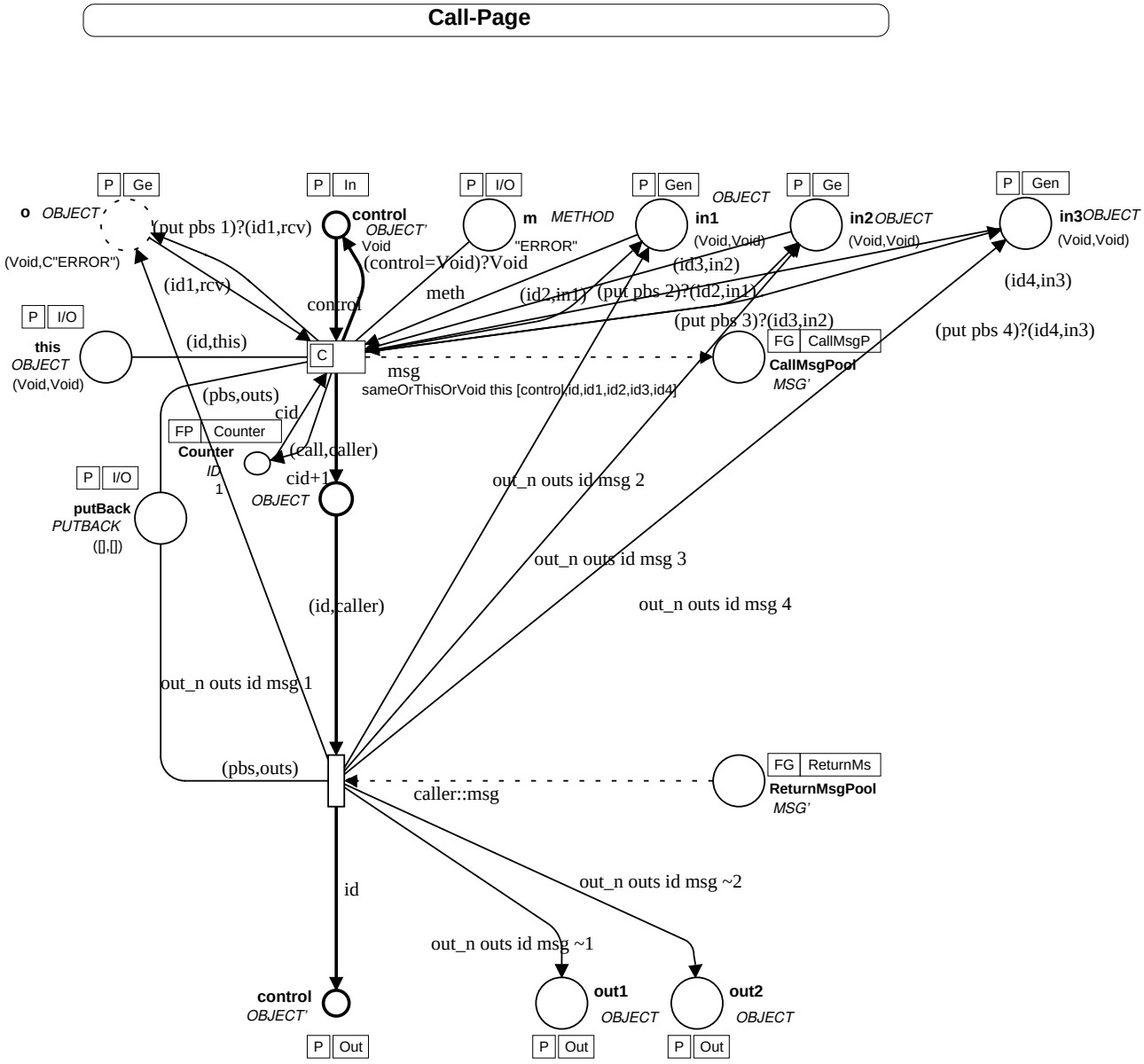
```
action
 let val name = "BusProcess";
  val (inf,outf) = my_execute("/home/tgi_3/tgi3/bin/messenger",[name]);
 in (my_pid := name; infile := inf; outfile := outf) end;
```




```
input (strmsg);
output (callmsgs,returnmsgs,stop);
action
 case toLocal (!my_pid) (strToMsg (strmsg)) of
    M(m)::rcv::param => (1‘(rcv::M(m)::param), empty, false)
  | I(i)::msg        => (empty,        1‘(I(i)::msg), false)
  | msg              => (empty,        1‘msg,         true);
```

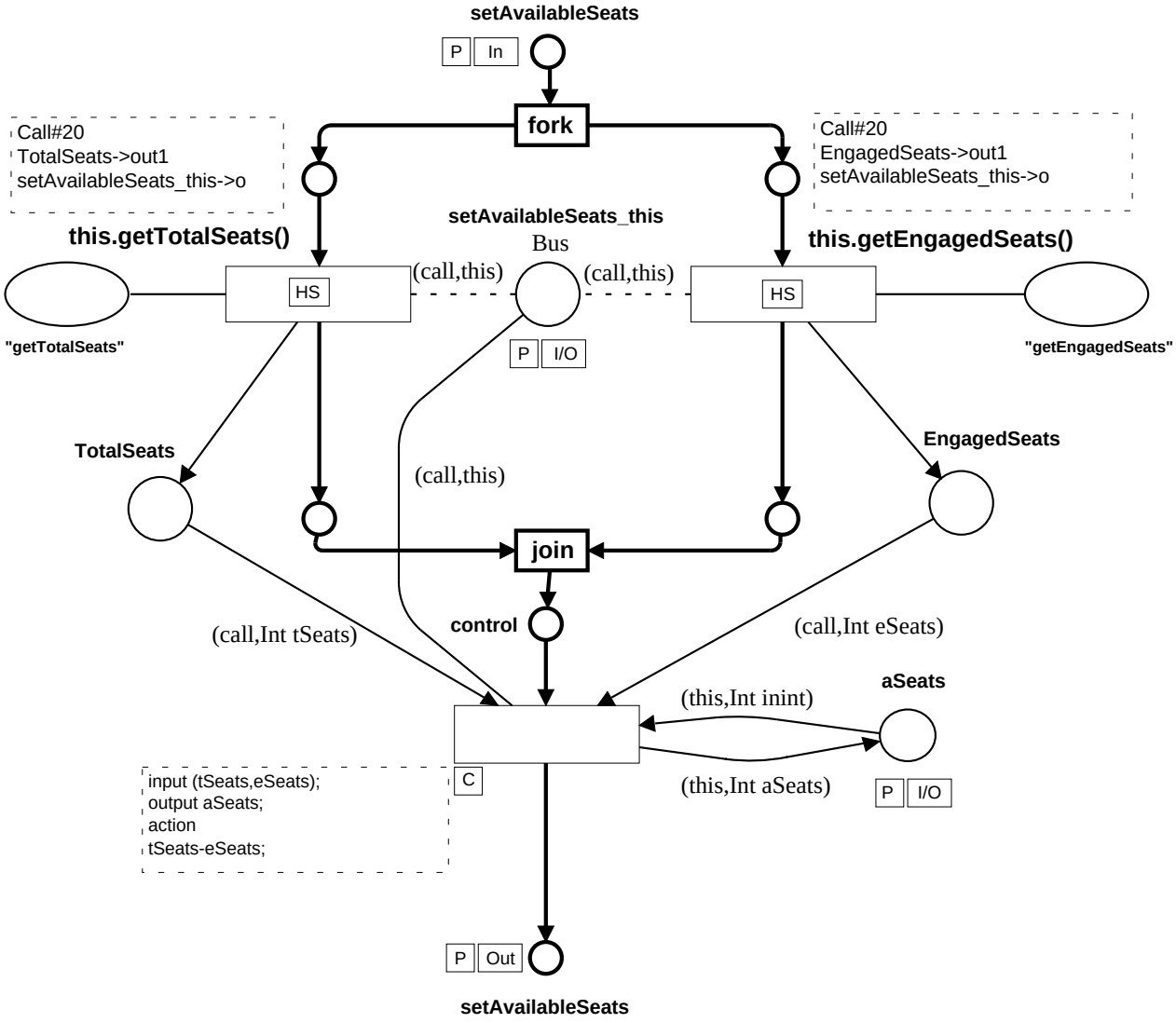
Class-Page

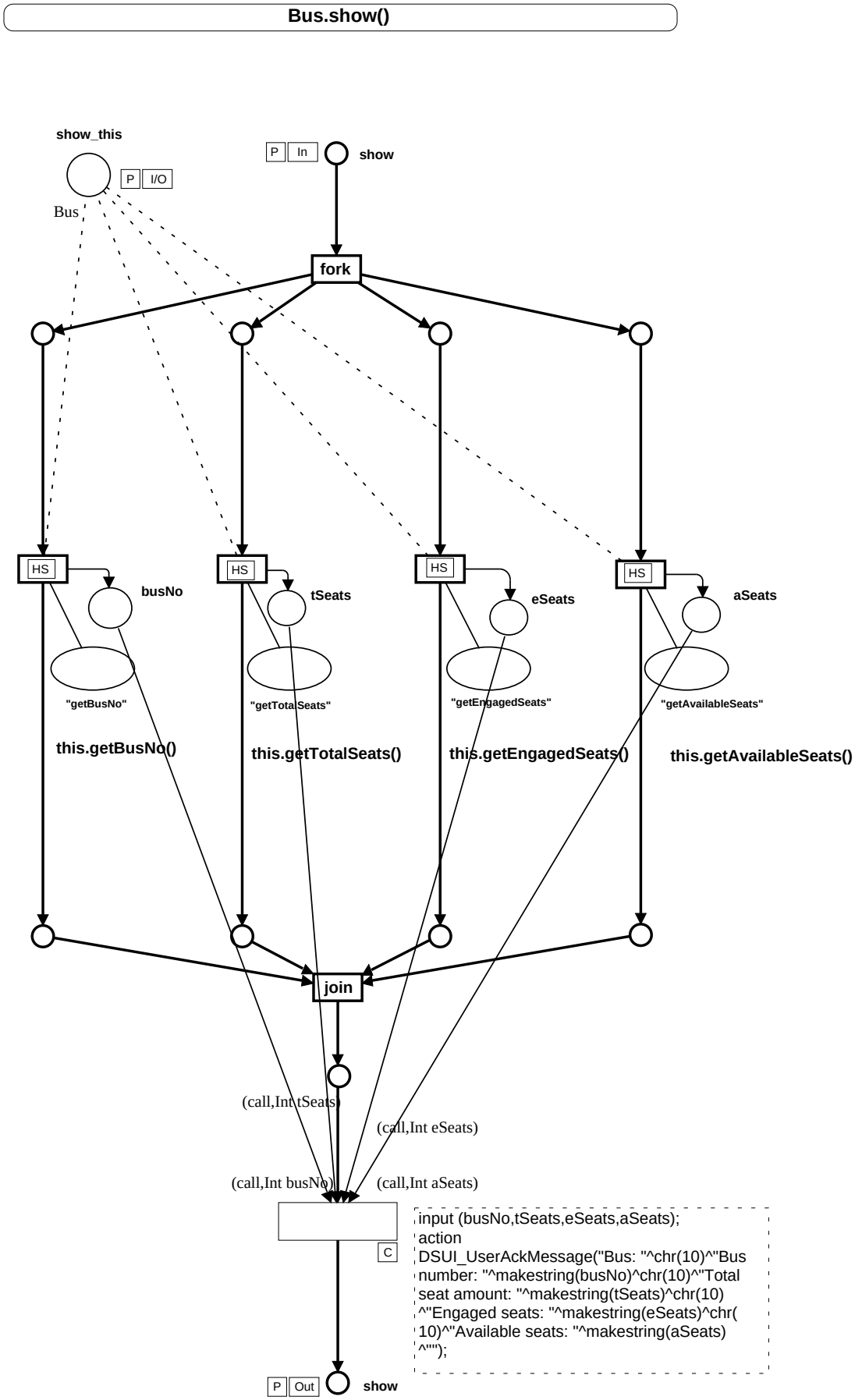

FG
CallMsgP
MSG'
P
I/O
Class
CLASS
ID
1
InstanceCounter
class
oid
oid+1
C class::M"new"::call::msg
(call,I(class,oid)::M"new"::msg)
class
I(class,oid)::M meth::call::msg
(call,I(class,oid)::M meth::msg)
class
[meth <> "new"]
C class::M meth::call::msg
(call,C class::M meth::msg)
P
Out
inMsg
MSG



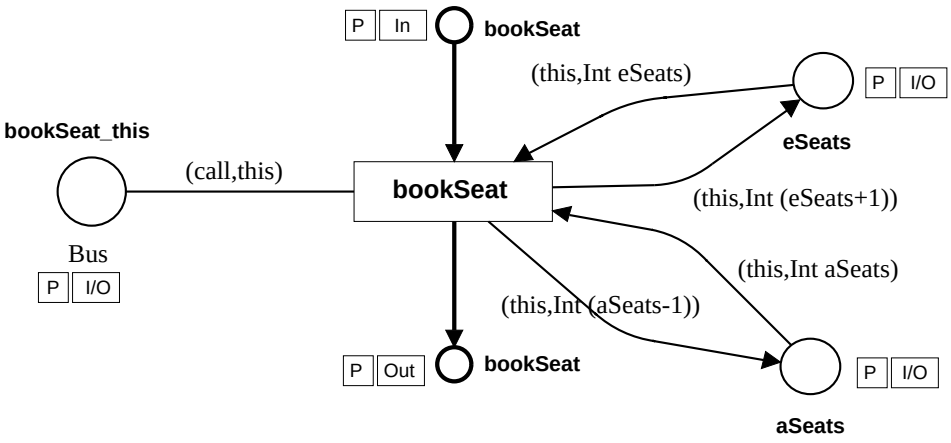

```
input (control,id,this,id1,rcv,id2,meth,id3,in1,in2,cid,in3,id4);
output(call,caller,msg);
action
  let val cntrl = nonThisOrVoid this [control,id,id1,id2,id3,id4];
    val caller = I("Call",cid);
in (cntrl,caller,rcv::M meth::caller::(noVoid [in1,in2,in3]))
end;
```

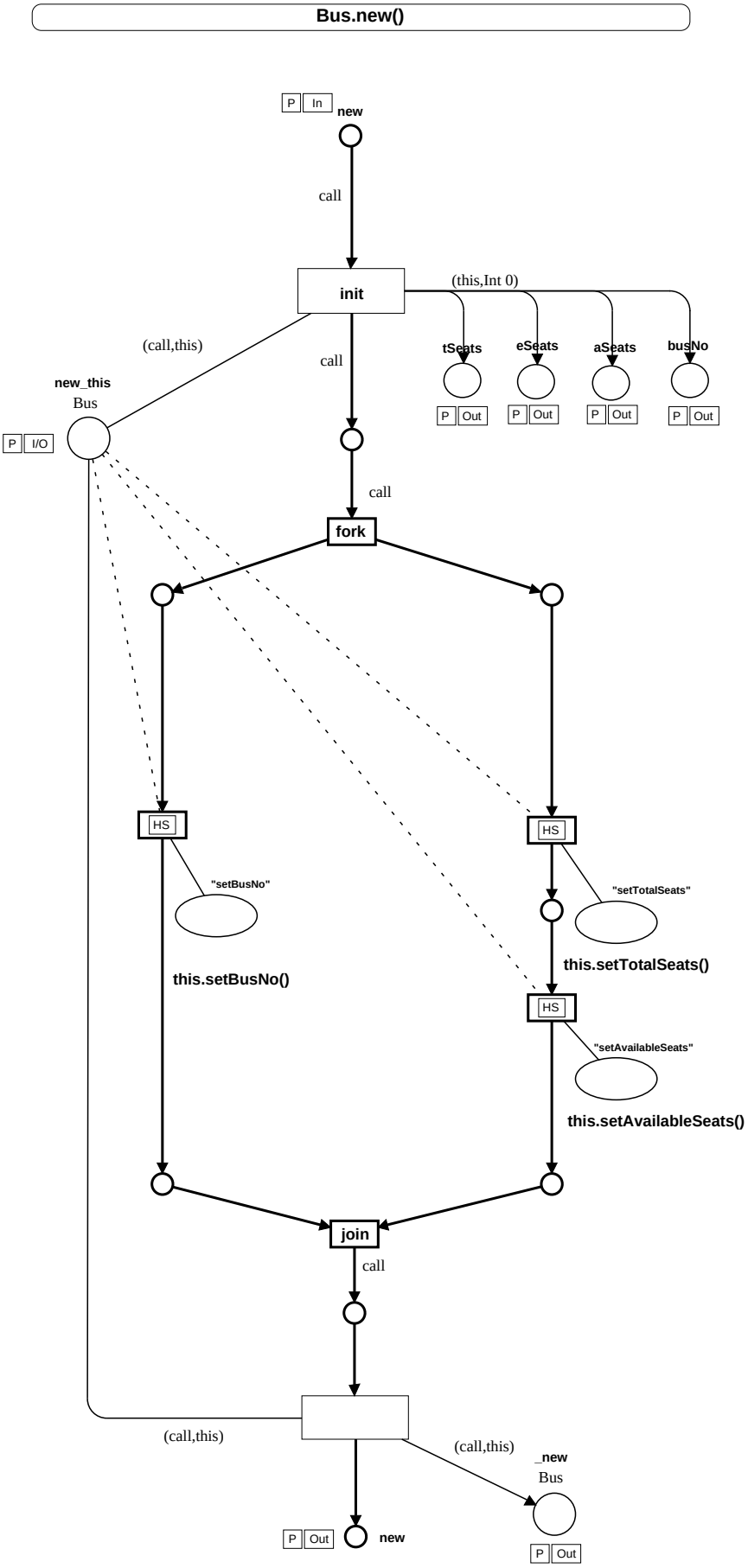
Bus.setAvailableSeats()

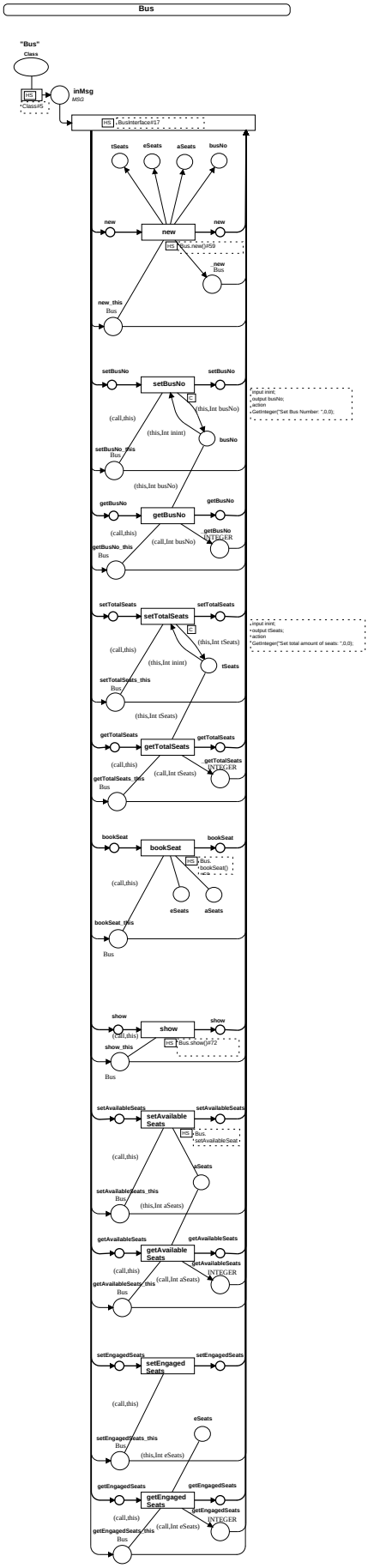


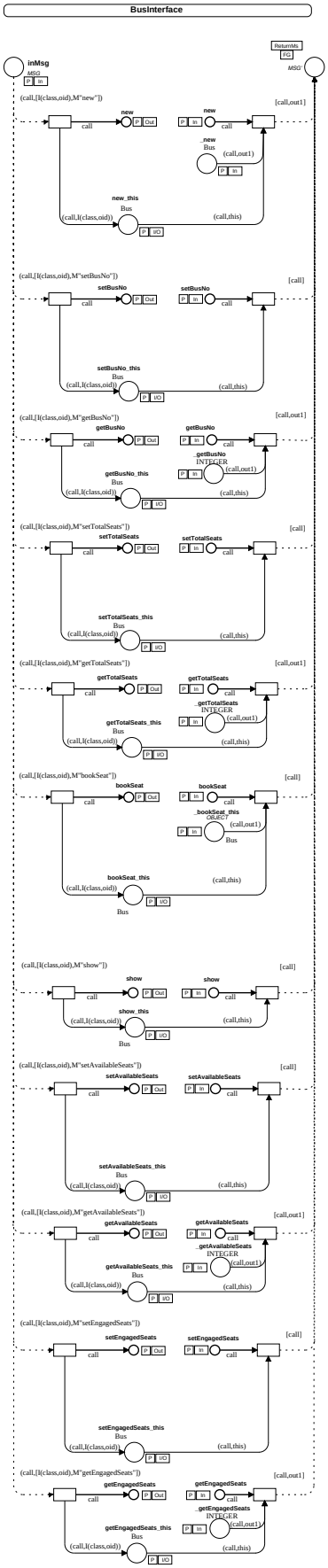

Bus.show()
show_this
P I/O
Bus
P In
show
fork
HS
busNo
tSeats
eSeats
aSeats
"getBusNo"
"getTotalSeats"
"getEngagedSeats"
"getAvailableSeats"
this.getBusNo()
this.getTotalSeats()
this.getEngagedSeats()
this.getAvailableSeats()
join
(call,Int tSeats)
(call,Int eSeats)
(call,Int busNo)
(call,Int aSeats)
C
input (busNo,tSeats,eSeats,aSeats);
action
DSUI_UserAckMessage("Bus: "^chr(10)^"Bus number: "^makestring(busNo)^chr(10)^"Total seat amount: "^makestring(tSeats)^chr(10)^"Engaged seats: "^makestring(eSeats)^chr(10)^"Available seats: "^makestring(aSeats)^"");
P Out
show

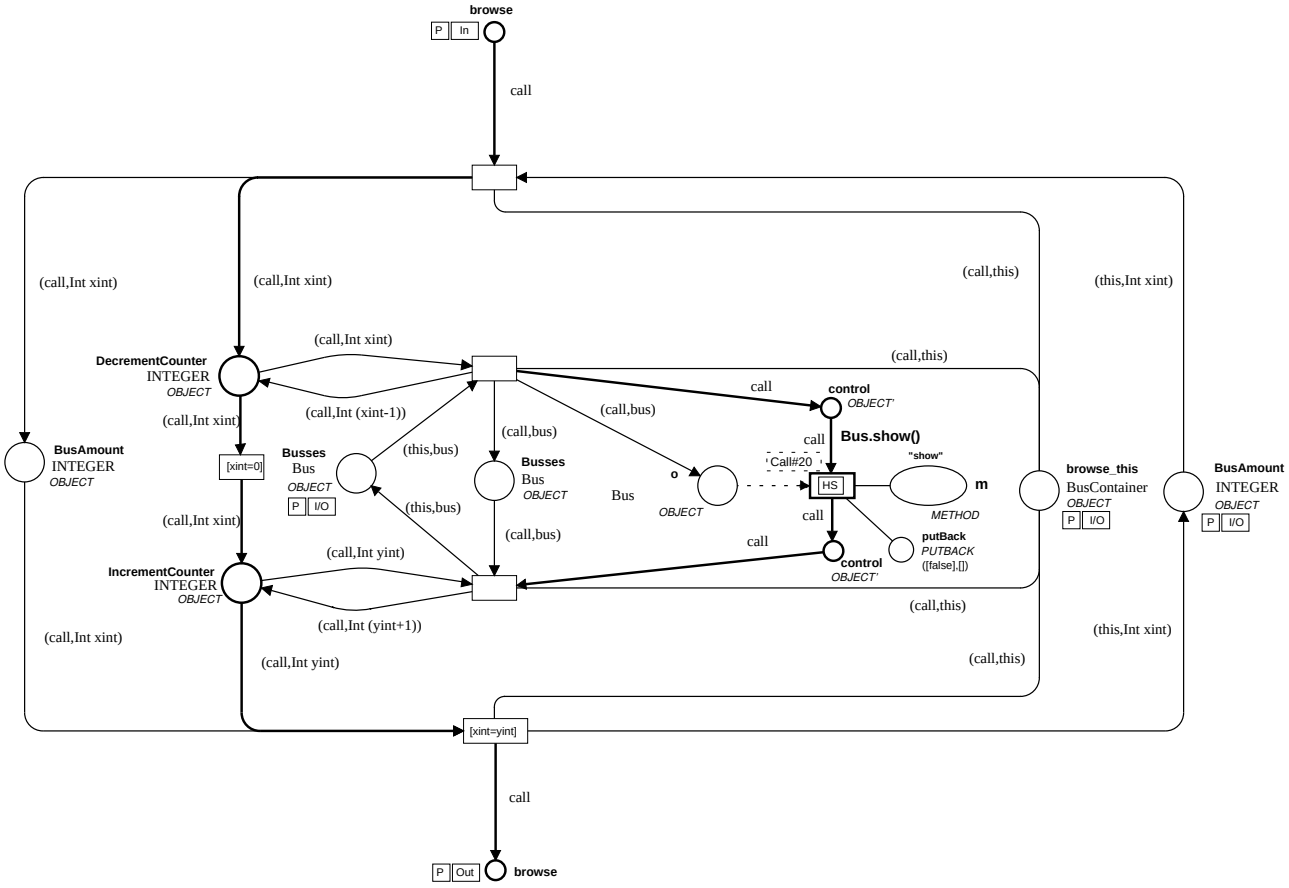
Bus.bookSeat()



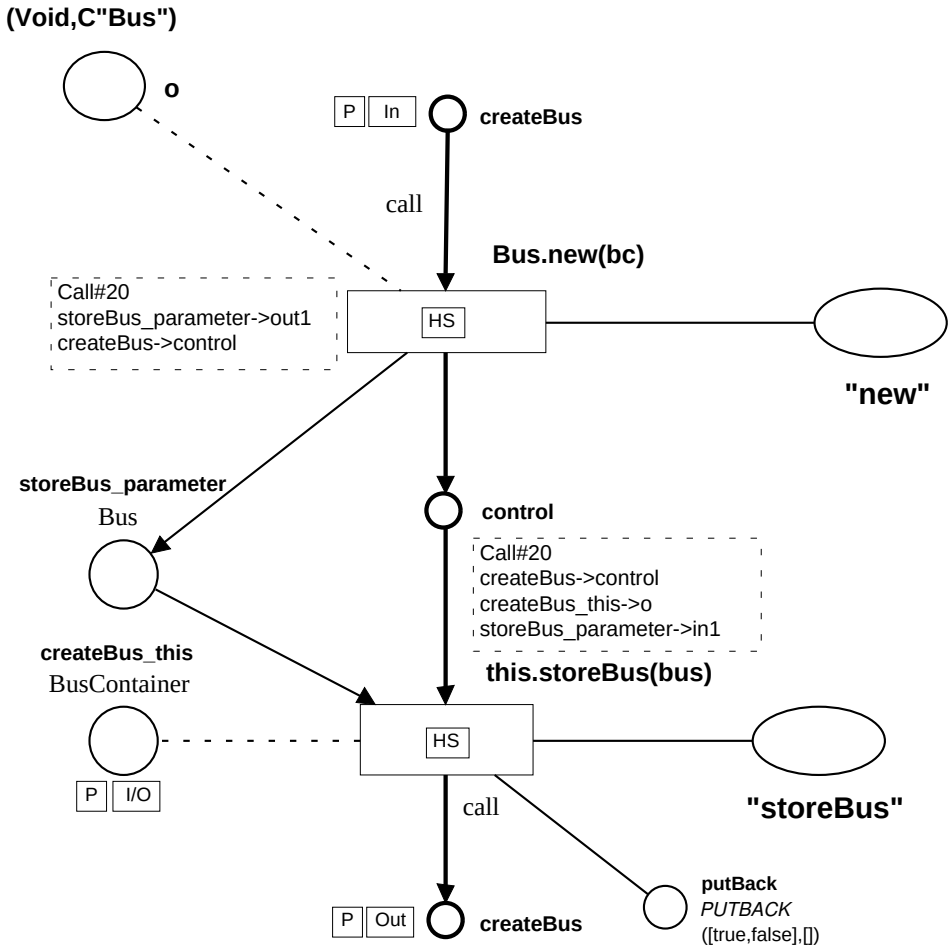

Bus.new()
P In
new
call
init
(this,Int 0)
tSeats
eSeats
aSeats
busNo
P Out
(call,this)
call
new_this
Bus
P I/O
call
fork
HS
"setBusNo"
this.setBusNo()
HS
"setTotalSeats"
this.setTotalSeats()
HS
"setAvailableSeats"
this.setAvailableSeats()
join
call
(call,this)
(call,this)
_new
Bus
P Out
new


Bus
"Bus"
Class
inMsg
MSG
HS
Class#5
BusInterface#17
tSeats
eSeats
aSeats
busNo
new
Bus.new()#59
setBusNo
(this,Int busNo)
(call,this)
(this,Int inint)
getBusNo
(call,Int busNo)
INTEGER
setTotalSeats
(this,Int tSeats)
getTotalSeats
(call,Int tSeats)
bookSeat
Bus.
bookSeat()
show
Bus.show()#72
setAvailableSeats
(this,Int aSeats)
getAvailableSeats
setEngagedSeats
(this,Int eSeats)
getEngagedSeats

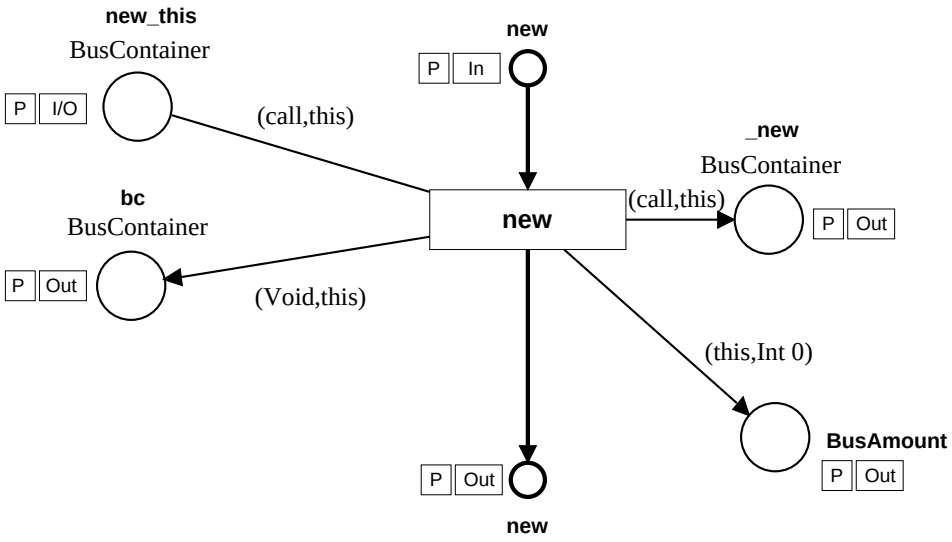

BusInterface
inMsg
MSG
P In
ReturnMs
FG
MSG'
(call,[I(class,oid),M"new"])
[call,out1]
new
call
P Out
P In
_new
Bus
(call,out1)
new_this
(call,I(class,oid))
(call,this)
P I/O
(call,[I(class,oid),M"setBusNo"])
[call]
setBusNo
setBusNo_this
(call,[I(class,oid),M"getBusNo"])
getBusNo
_getBusNo
INTEGER
getBusNo_this
(call,[I(class,oid),M"setTotalSeats"])
setTotalSeats
setTotalSeats_this
(call,[I(class,oid),M"getTotalSeats"])
getTotalSeats
_getTotalSeats
getTotalSeats_this
(call,[I(class,oid),M"bookSeat"])
bookSeat
_bookSeat_this
OBJECT
bookSeat_this
Bus
(call,[I(class,oid),M"show"])
show
show_this
(call,[I(class,oid),M"setAvailableSeats"])
setAvailableSeats
setAvailableSeats_this
(call,[I(class,oid),M"getAvailableSeats"])
getAvailableSeats
_getAvailableSeats
getAvailableSeats_this
(call,[I(class,oid),M"setEngagedSeats"])
setEngagedSeats
setEngagedSeats_this
(call,[I(class,oid),M"getEngagedSeats"])
getEngagedSeats
_getEngagedSeats
getEngagedSeats_this


BusContainer.browse()
browse
P In
call
(call,Int xint)
(call,Int xint)
(call,this)
(this,Int xint)
DecrementCounter
INTEGER
OBJECT
(call,Int xint)
(call,this)
(call,Int (xint-1))
(call,bus)
call
control
OBJECT'
Bus.show()
BusAmount
INTEGER
OBJECT
[xint=0]
Busses
Bus
OBJECT
P I/O
(this,bus)
Busses
Bus
OBJECT
Bus
o
OBJECT
Call#20
HS
"show"
m
METHOD
browse_this
BusContainer
OBJECT
P I/O
BusAmount
INTEGER
OBJECT
P I/O
(call,Int xint)
(this,bus)
(call,bus)
call
putBack
PUTBACK
([false],[])
IncrementCounter
INTEGER
OBJECT
(call,Int yint)
control
OBJECT'
(call,this)
(call,Int (yint+1))
(call,Int xint)
(this,Int xint)
(call,Int yint)
(call,this)
[xint=yint]
call
P Out
browse

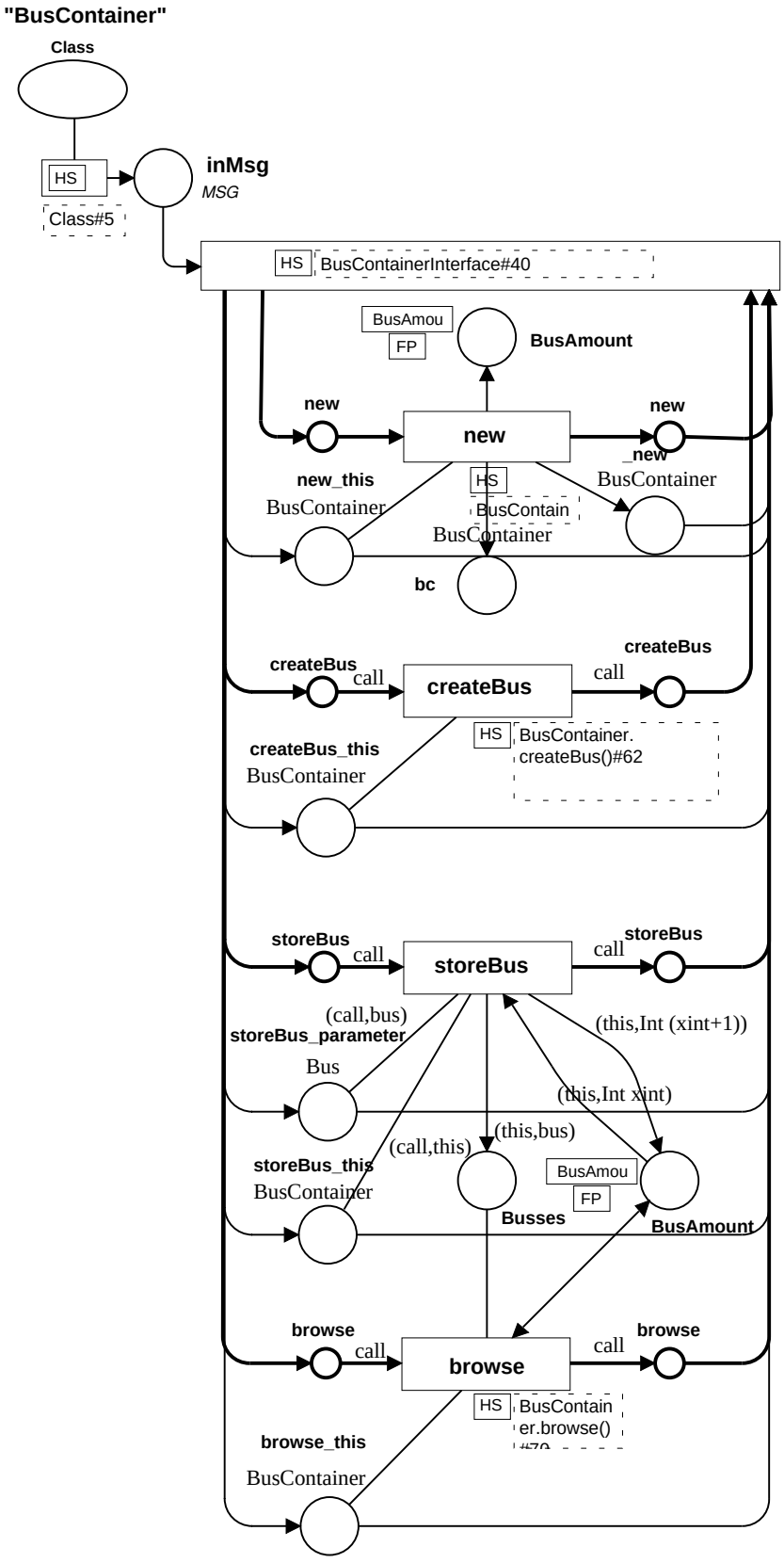
BusContainer.createBus()

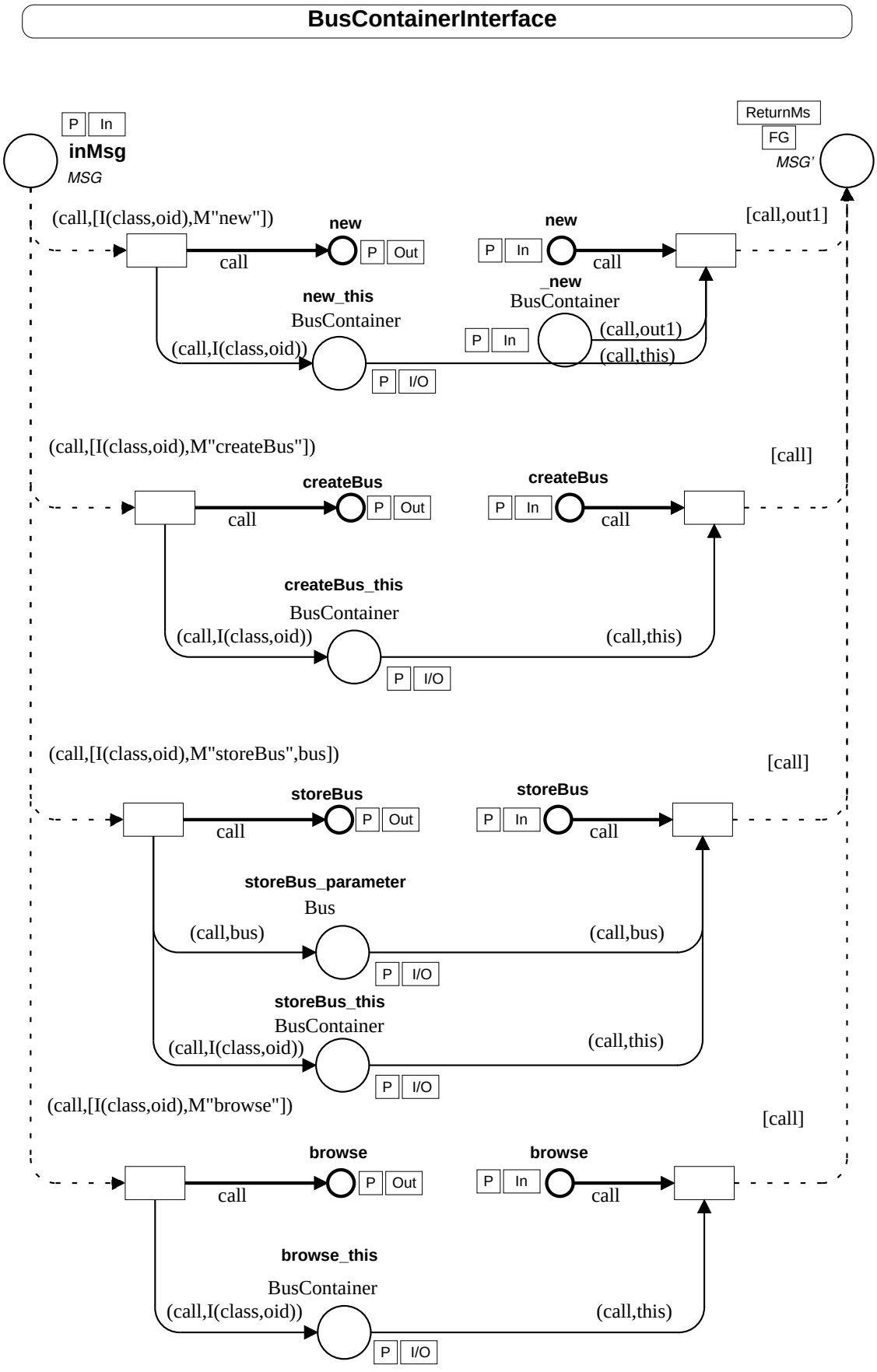


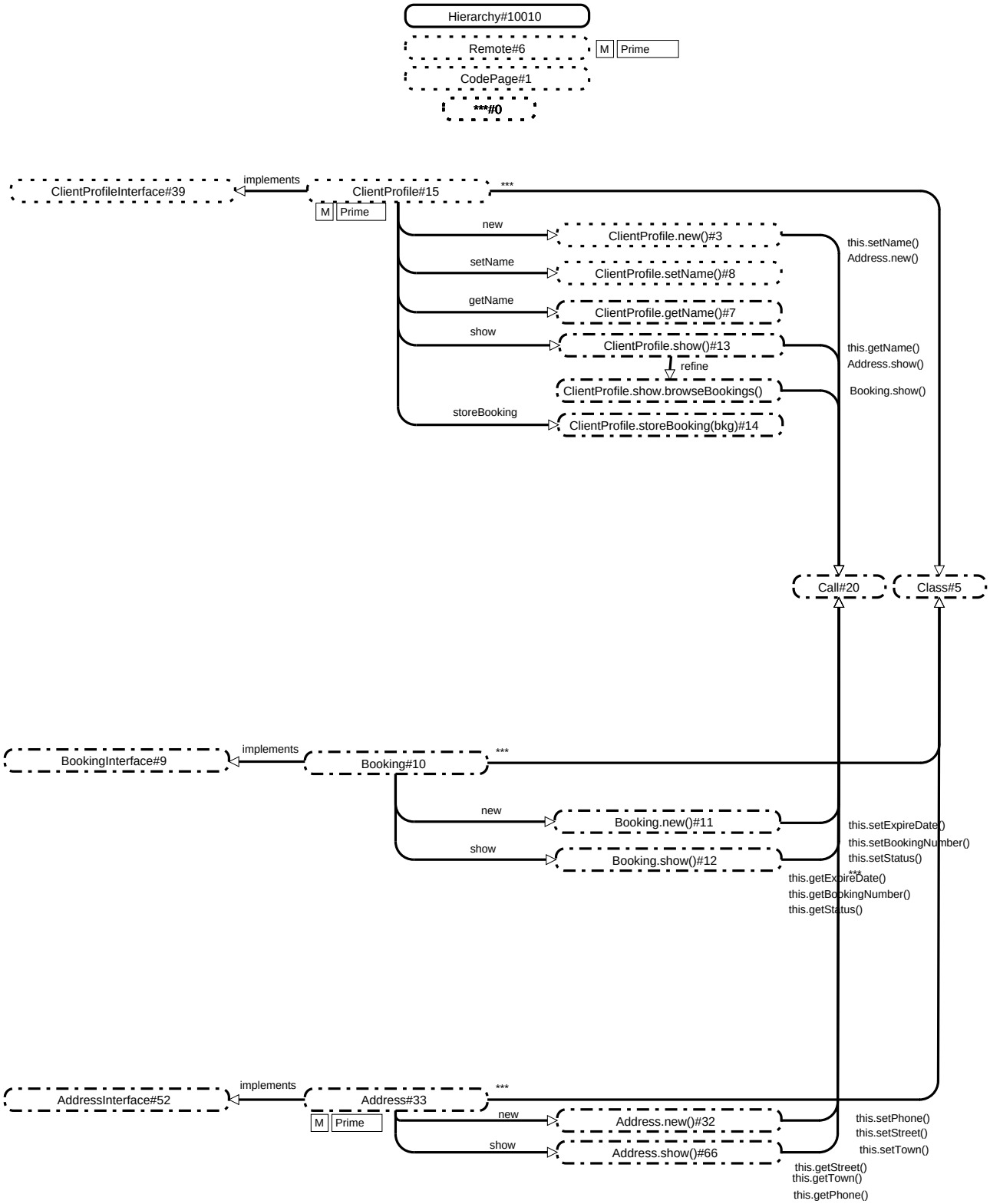
BusContainer.new()

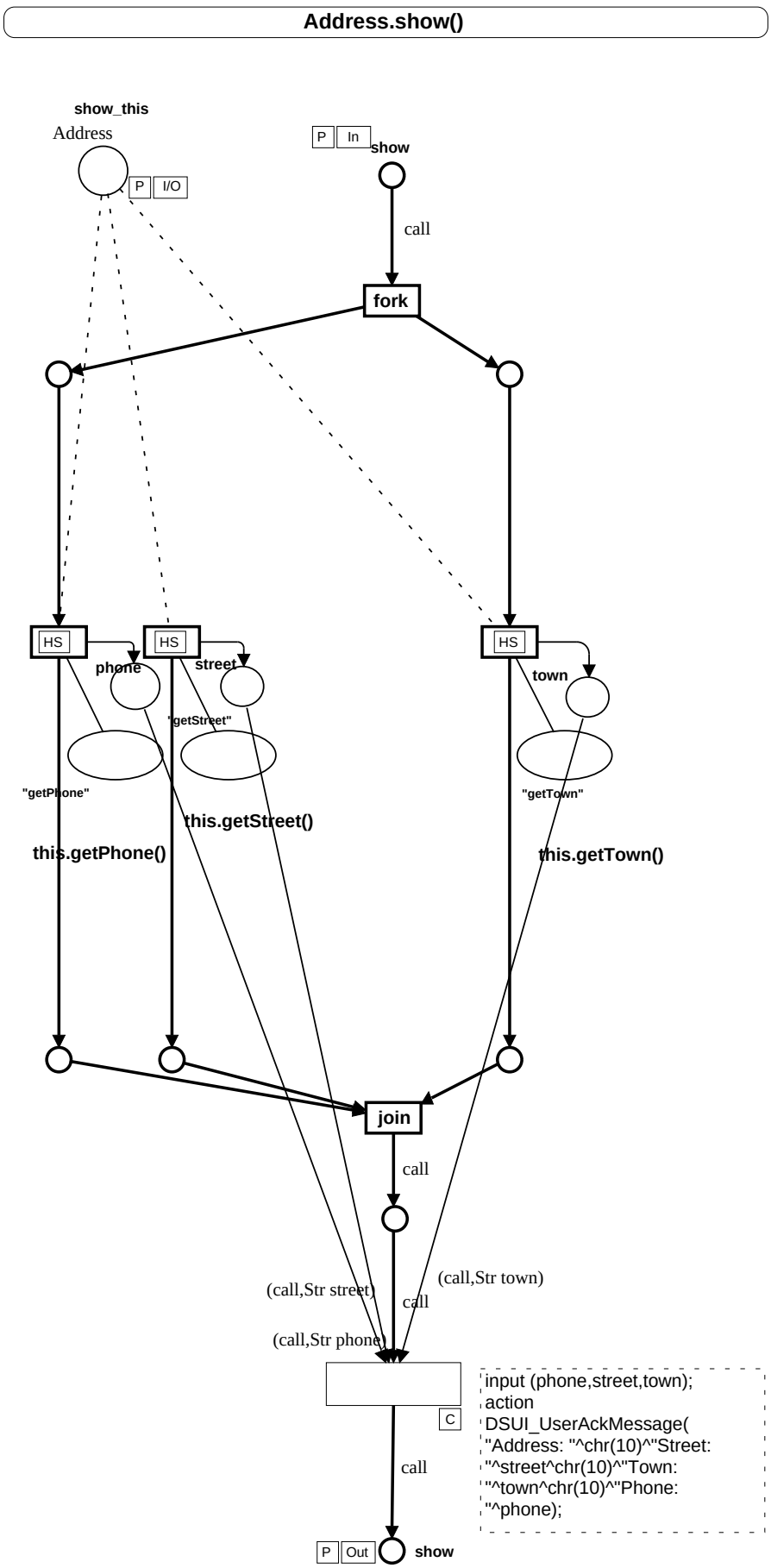


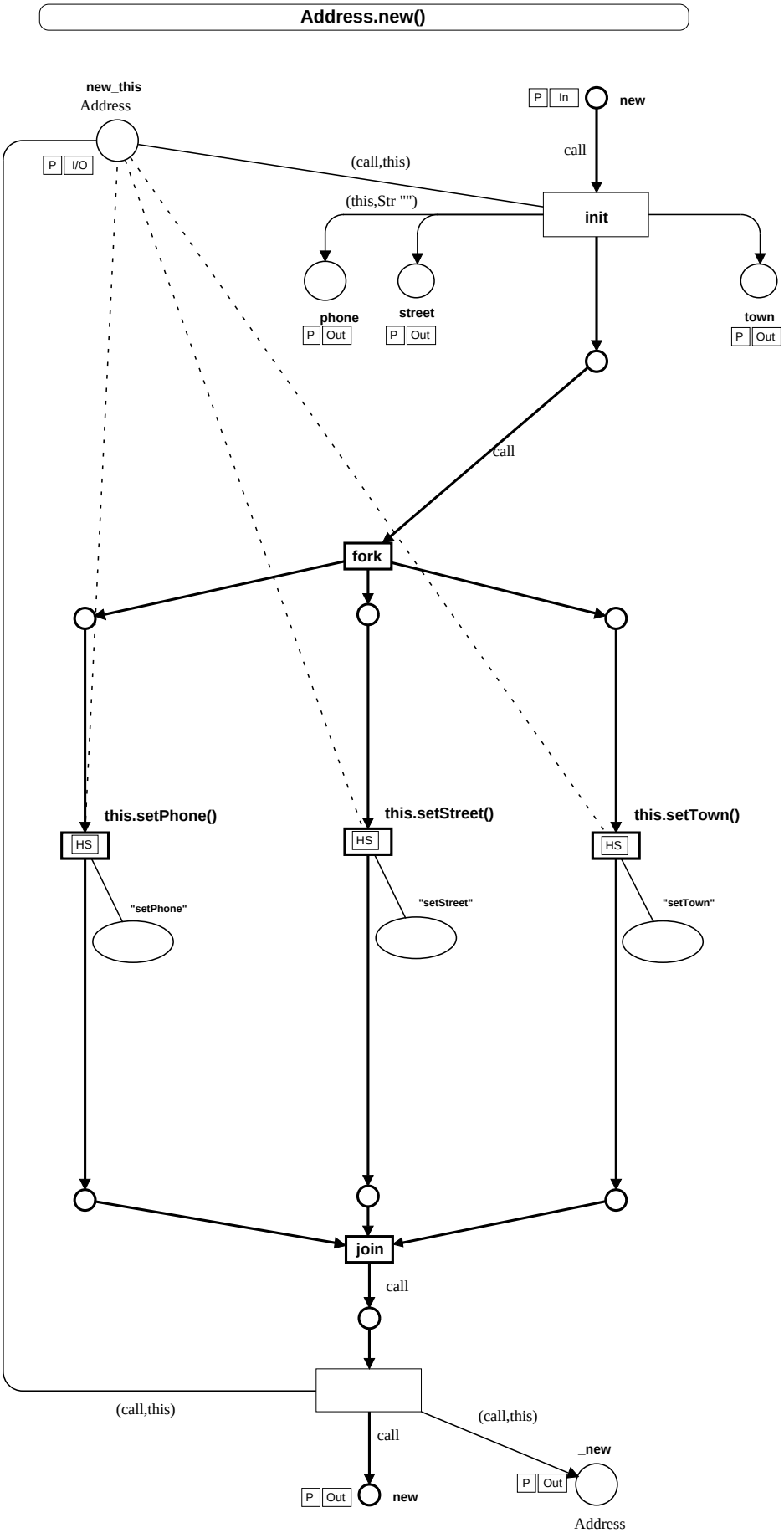
BusContainer

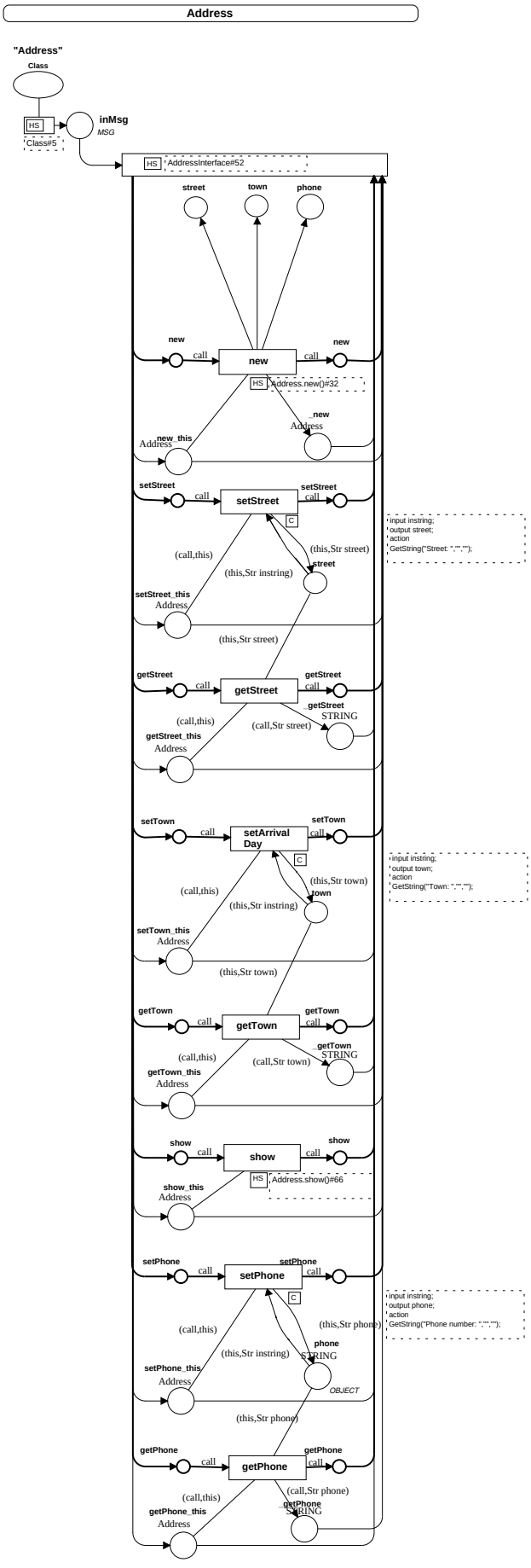


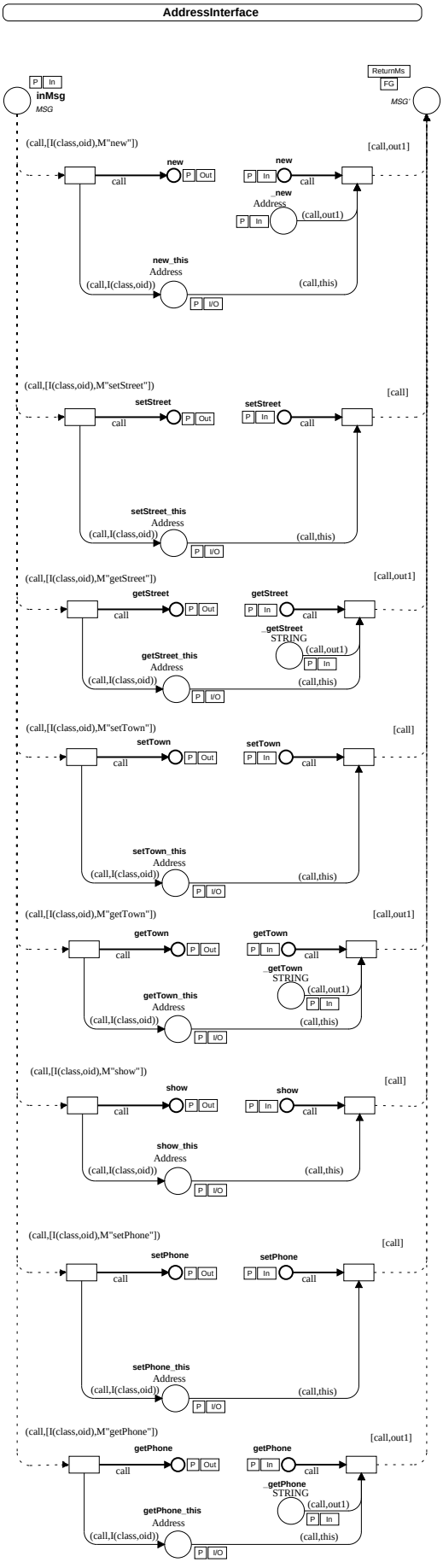

BusContainerInterface
P
In
inMsg
MSG
ReturnMs
FG
MSG'
(call,[I(class,oid),M"new"])
new
call
P
Out
new
P
In
call
[call,out1]
new_this
BusContainer
_new
BusContainer
(call,I(class,oid))
P
In
(call,out1)
(call,this)
P
I/O
(call,[I(class,oid),M"createBus"])
[call]
createBus
call
P
Out
createBus
P
In
call
createBus_this
BusContainer
(call,I(class,oid))
(call,this)
P
I/O
(call,[I(class,oid),M"storeBus",bus])
[call]
storeBus
call
P
Out
storeBus
P
In
call
storeBus_parameter
Bus
(call,bus)
(call,bus)
P
I/O
storeBus_this
BusContainer
(call,I(class,oid))
(call,this)
P
I/O
(call,[I(class,oid),M"browse"])
[call]
browse
call
P
Out
browse
P
In
call
browse_this
BusContainer
(call,I(class,oid))
(call,this)
P
I/O


Hierarchy#10010
Remote#6
M
Prime
CodePage#1
***#0
ClientProfileInterface#39
implements
ClientProfile#15
***
M
Prime
new
ClientProfile.new()#3
this.setName()
Address.new()
setName
ClientProfile.setName()#8
getName
ClientProfile.getName()#7
show
ClientProfile.show()#13
this.getName()
Address.show()
refine
ClientProfile.show.browseBookings()
Booking.show()
storeBooking
ClientProfile.storeBooking(bkg)#14
Call#20
Class#5
BookingInterface#9
implements
Booking#10
***
new
Booking.new()#11
this.setExpireDate()
this.setBookingNumber()
this.setStatus()
show
Booking.show()#12
***
this.getExpireDate()
this.getBookingNumber()
this.getStatus()
AddressInterface#52
implements
Address#33
***
M
Prime
new
Address.new()#32
this.setPhone()
this.setStreet()
this.setTown()
show
Address.show()#66
this.getStreet()
this.getTown()
this.getPhone()

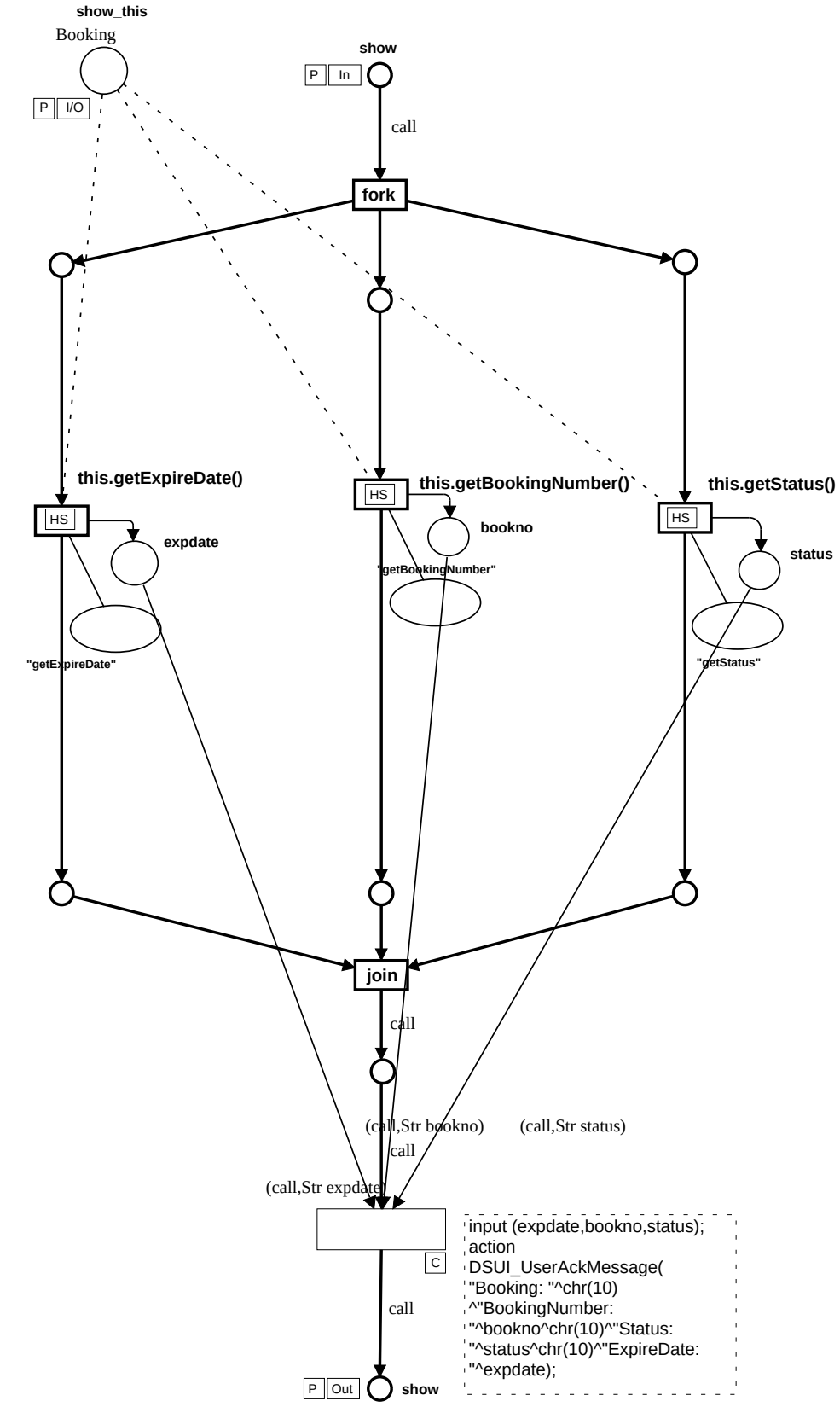

Address.show()
show_this
Address
P
I/O
P
In
show
call
fork
HS
HS
HS
phone
street
town
"getStreet"
"getPhone"
"getTown"
this.getStreet()
this.getPhone()
this.getTown()
join
call
(call,Str street)
(call,Str town)
call
(call,Str phone)
C
call
P
Out
show
input (phone,street,town);
action
DSUI_UserAckMessage(
"Address: "^chr(10)^"Street:
"^street^chr(10)^"Town:
"^town^chr(10)^"Phone:
"^phone);

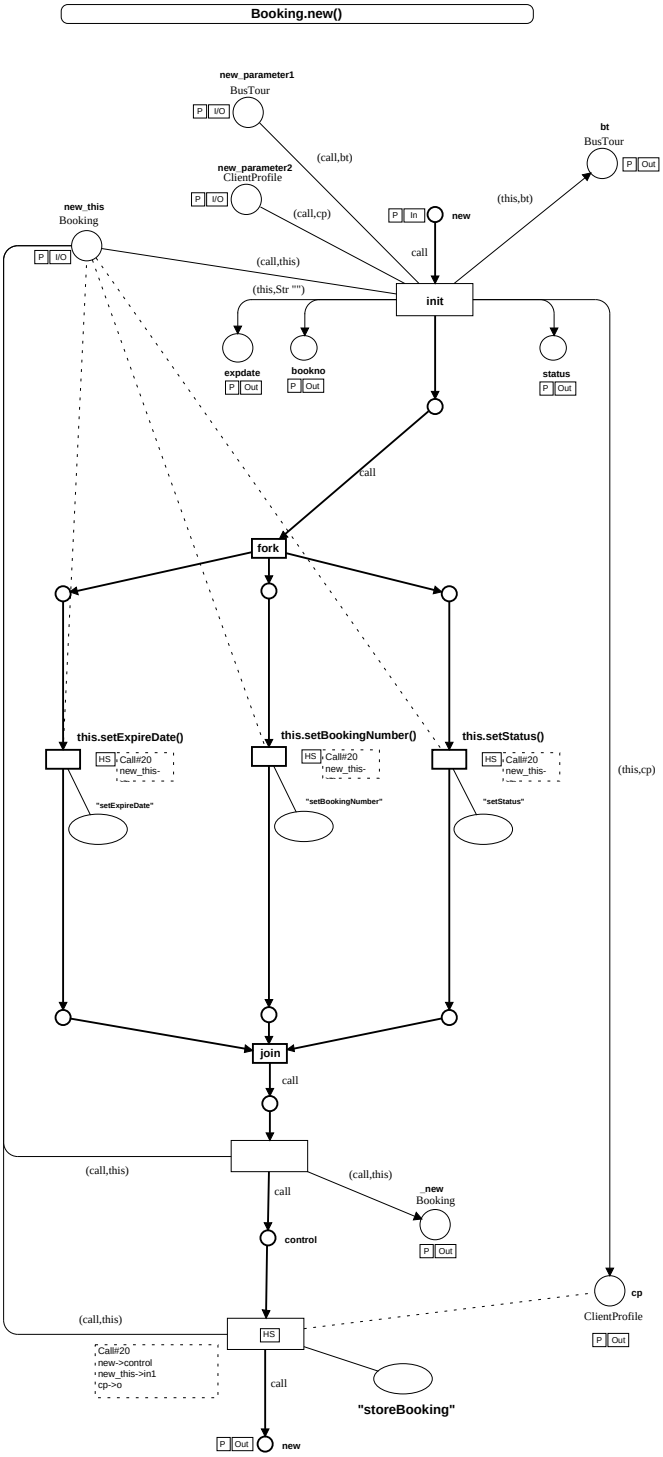

Address.new()
new_this
Address
P I/O
P In
new
call
(call,this)
(this,Str "")
init
phone
P Out
street
P Out
town
P Out
call
fork
this.setPhone()
HS
"setPhone"
this.setStreet()
HS
"setStreet"
this.setTown()
HS
"setTown"
join
call
(call,this)
call
(call,this)
_new
P Out
P Out
new
Address


Address
"Address"
Class
HS
Class#5
inMsg
MSG
HS
AddressInterface#52
street
town
phone
new
call
new
HS
Address.new()#32
_new
Address
new_this
Address
setStreet
call
setStreet
C
input instring;
output street;
action
GetString("Street: ","","");
(call,this)
(this,Str street)
street
(this,Str instring)
setStreet_this
Address
(this,Str street)
getStreet
call
getStreet
_getStreet
STRING
(call,this)
(call,Str street)
getStreet_this
Address
setTown
call
setArrival Day
setTown
C
input instring;
output town;
action
GetString("Town: ","","");
(this,Str town)
town
(call,this)
(this,Str instring)
setTown_this
Address
(this,Str town)
getTown
call
getTown
_getTown
STRING
(call,this)
(call,Str town)
getTown_this
Address
show
call
show
HS
Address.show()#66
show_this
Address
setPhone
call
setPhone
C
input instring;
output phone;
action
GetString("Phone number: ","","");
(call,this)
(this,Str phone)
phone
STRING
(this,Str instring)
setPhone_this
Address
OBJECT
(this,Str phone)
getPhone
call
getPhone
(call,this)
(call,Str phone)
getPhone
STRING
getPhone_this
Address

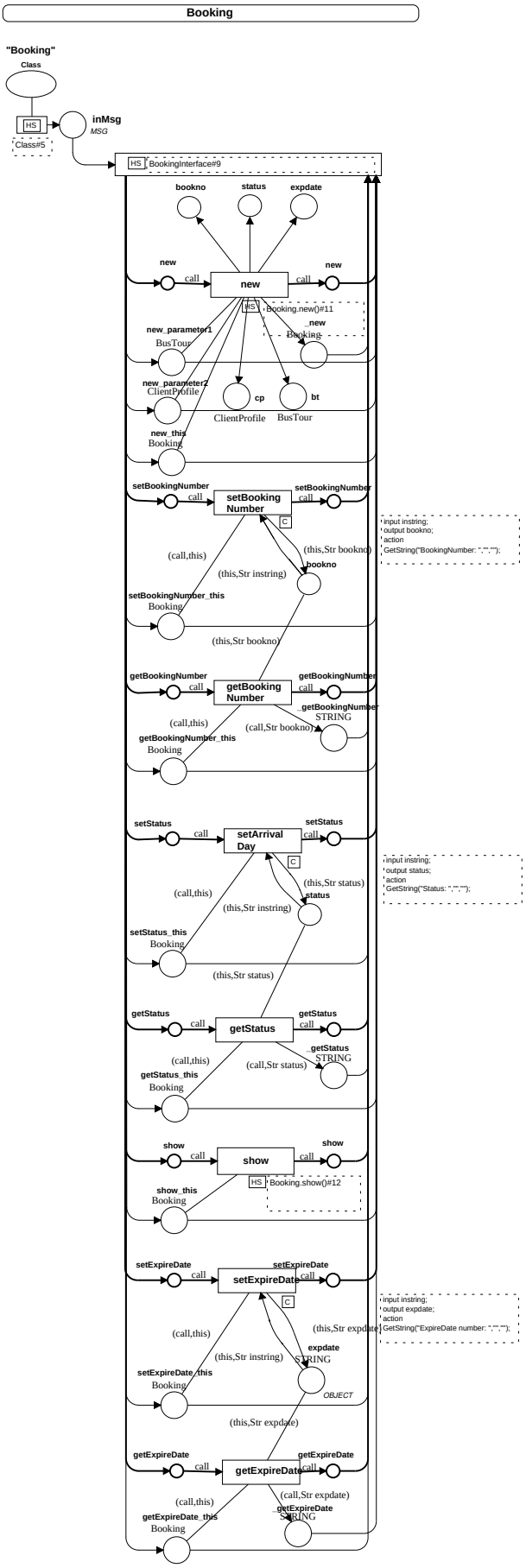

AddressInterface
P In
inMsg
MSG
ReturnMs
FG
MSG'
(call,[I(class,oid),M"new"])
[call,out1]
new
call
P Out
P In
_new
Address
(call,out1)
new_this
Address
(call,I(class,oid))
(call,this)
P I/O
(call,[I(class,oid),M"setStreet"])
[call]
setStreet
setStreet_this
(call,[I(class,oid),M"getStreet"])
[call,out1]
getStreet
_getStreet
STRING
getStreet_this
(call,[I(class,oid),M"setTown"])
[call]
setTown
setTown_this
(call,[I(class,oid),M"getTown"])
[call,out1]
getTown
_getTown
STRING
getTown_this
(call,[I(class,oid),M"show"])
[call]
show
show_this
(call,[I(class,oid),M"setPhone"])
[call]
setPhone
setPhone_this
(call,[I(class,oid),M"getPhone"])
[call,out1]
getPhone
_getPhone
STRING
getPhone_this

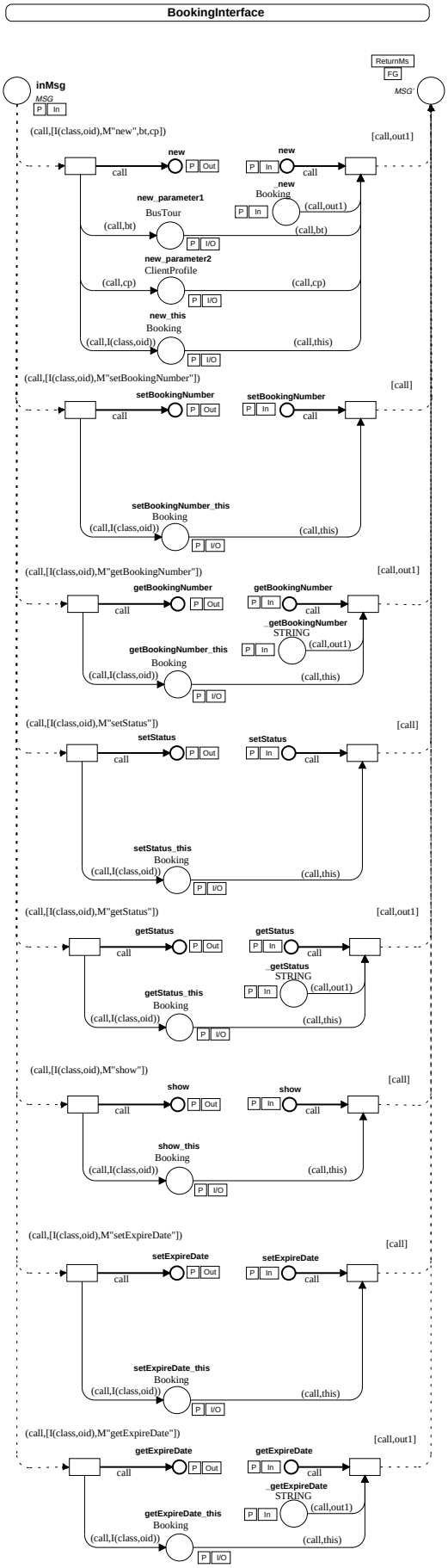
Booking.show()



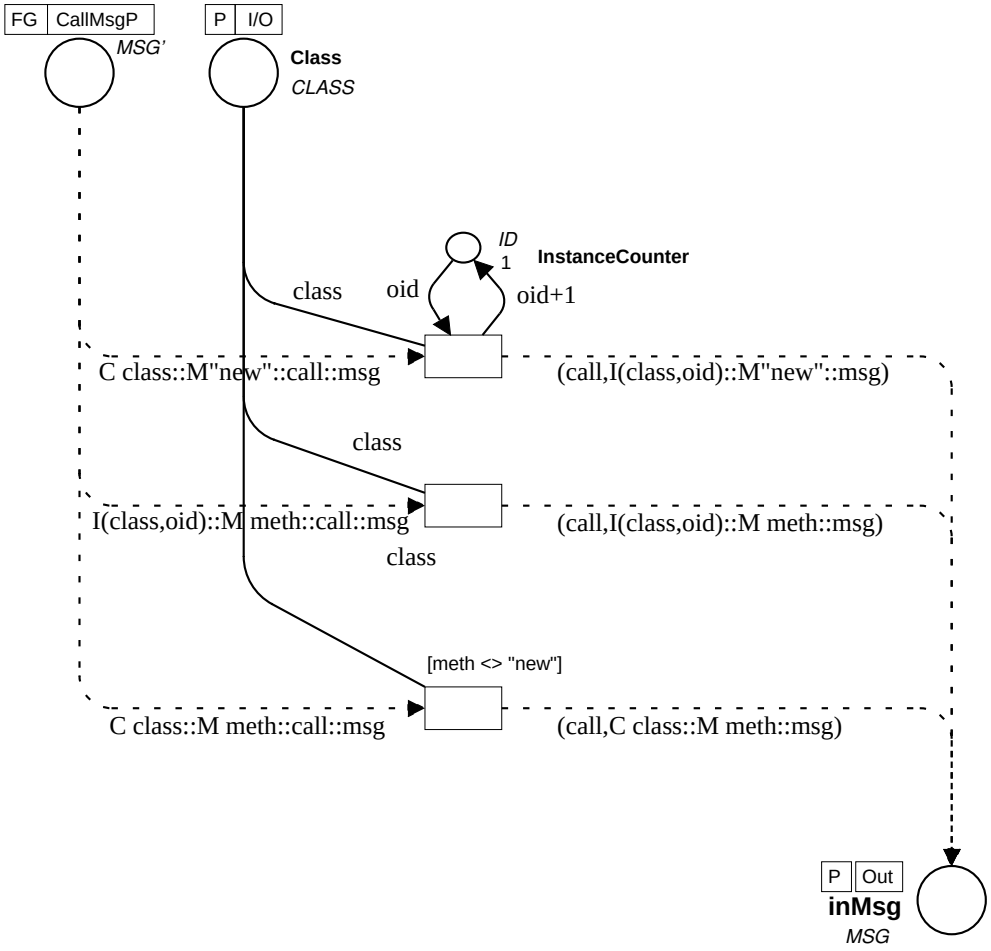

Booking.new()
new_parameter1
BusTour
P I/O
new_parameter2
ClientProfile
P I/O
new_this
Booking
P I/O
(call,bt)
(call,cp)
(call,this)
(this,Str "")
P In
new
call
init
(this,bt)
bt
BusTour
P Out
expdate
P Out
bookno
P Out
status
P Out
call
fork
this.setExpireDate()
HS Call#20 new_this-
"setExpireDate"
this.setBookingNumber()
HS Call#20 new_this-
"setBookingNumber"
this.setStatus()
HS Call#20 new_this-
"setStatus"
(this,cp)
join
call
(call,this)
(call,this)
call
_new
Booking
P Out
control
(call,this)
HS
Call#20
new->control
new_this->in1
cp->0
call
"storeBooking"
P Out
new
cp
ClientProfile
P Out

(call,this)

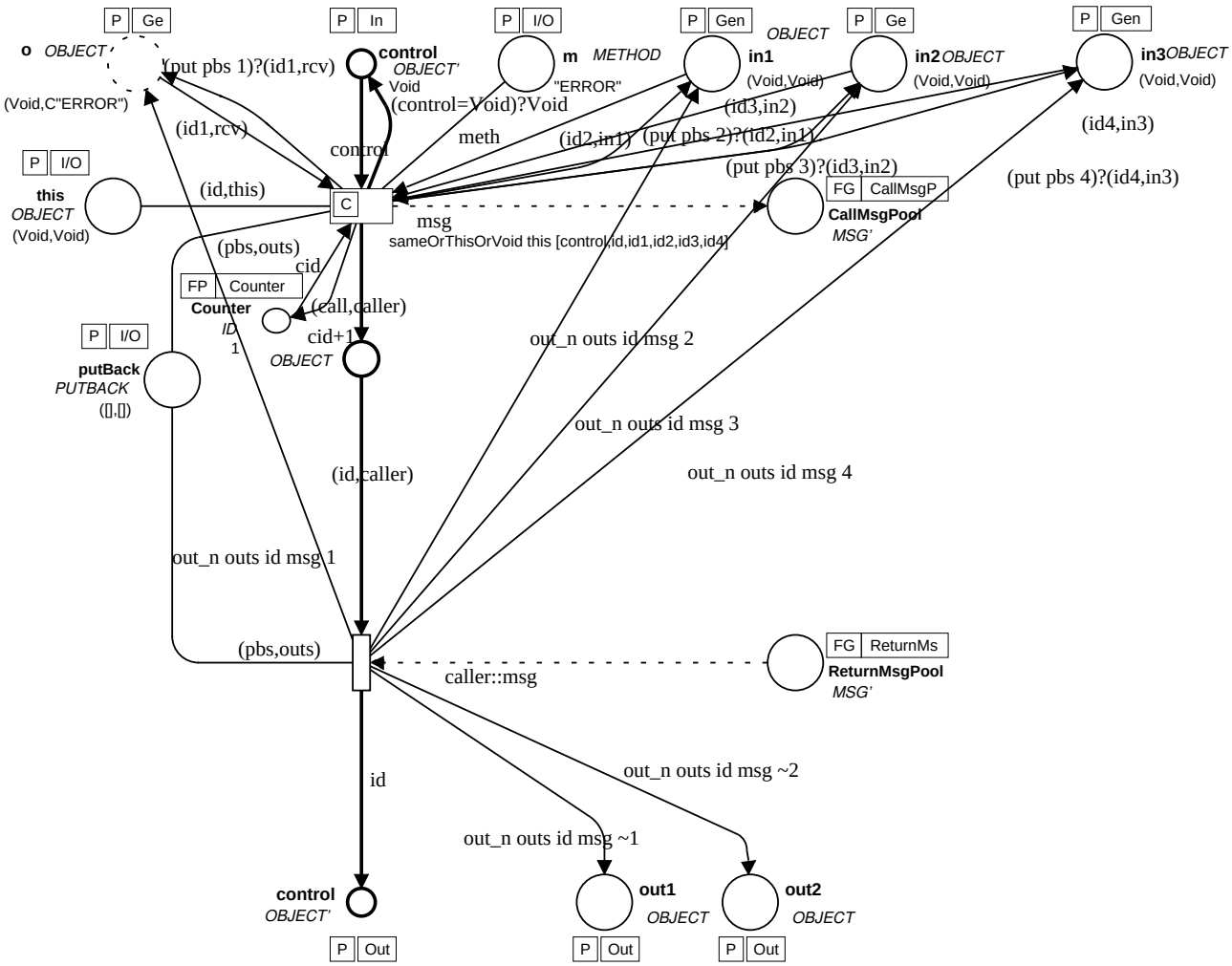

Booking
"Booking"
Class
HS
Class#5
inMsg
MSG
HS
BookingInterface#9
bookno
status
expdate
new
call
new
call
new
HS
Booking.new()#11
new_parameter1
BusTour
new
Booking
new_parameter2
ClientProfile
cp
bt
ClientProfile
BusTour
new_this
Booking
setBookingNumber
call
setBooking Number
setBookingNumber
call
C
input instring;
output bookno;
action
GetString("BookingNumber: ","","");
(call,this)
(this,Str bookno)
bookno
(this,Str instring)
setBookingNumber_this
Booking
(this,Str bookno)
getBookingNumber
call
getBooking Number
getBookingNumber
call
getBookingNumber
STRING
(call,this)
(call,Str bookno)
getBookingNumber_this
Booking
setStatus
call
setArrival Day
setStatus
call
C
input instring;
output status;
action
GetString("Status: ","","");
(call,this)
(this,Str status)
status
(this,Str instring)
setStatus_this
Booking
(this,Str status)
getStatus
call
getStatus
getStatus
call
getStatus
STRING
(call,this)
(call,Str status)
getStatus_this
Booking
show
call
show
show
call
HS
Booking.show()#12
show_this
Booking
setExpireDate
call
setExpireDate
setExpireDate
call
C
input instring;
output expdate;
action
GetString("ExpireDate number: ","","");
(call,this)
(this,Str expdate)
expdate
STRING
(this,Str instring)
setExpireDate_this
Booking
OBJECT
(this,Str expdate)
getExpireDate
call
getExpireDate
getExpireDate
call
(call,Str expdate)
(call,this)
getExpireDate
STRING
getExpireDate_this
Booking


BookingInterface
inMsg
MSG
P In
ReturnMs
FG
MSG'
(call,[I(class,oid),M"new",bt,cp])
[call,out1]
new
P Out
call
new
P In
call
_new
Booking
P In
(call,out1)
new_parameter1
BusTour
(call,bt)
P I/O
(call,bt)
new_parameter2
ClientProfile
(call,cp)
P I/O
(call,cp)
new_this
Booking
(call,I(class,oid))
P I/O
(call,this)
(call,[I(class,oid),M"setBookingNumber"])
[call]
setBookingNumber
P Out
call
setBookingNumber
P In
call
setBookingNumber_this
Booking
(call,I(class,oid))
P I/O
(call,this)
(call,[I(class,oid),M"getBookingNumber"])
[call,out1]
getBookingNumber
P Out
call
getBookingNumber
P In
call
_getBookingNumber
STRING
P In
(call,out1)
getBookingNumber_this
Booking
(call,I(class,oid))
P I/O
(call,this)
(call,[I(class,oid),M"setStatus"])
[call]
setStatus
P Out
call
setStatus
P In
call
setStatus_this
Booking
(call,I(class,oid))
P I/O
(call,this)
(call,[I(class,oid),M"getStatus"])
[call,out1]
getStatus
P Out
call
getStatus
P In
call
_getStatus
STRING
P In
(call,out1)
getStatus_this
Booking
(call,I(class,oid))
P I/O
(call,this)
(call,[I(class,oid),M"show"])
[call]
show
P Out
call
show
P In
call
show_this
Booking
(call,I(class,oid))
P I/O
(call,this)
(call,[I(class,oid),M"setExpireDate"])
[call]
setExpireDate
P Out
call
setExpireDate
P In
call
setExpireDate_this
Booking
(call,I(class,oid))
P I/O
(call,this)
(call,[I(class,oid),M"getExpireDate"])
[call,out1]
getExpireDate
P Out
call
getExpireDate
P In
call
_getExpireDate
STRING
P In
(call,out1)
getExpireDate_this
Booking
(call,I(class,oid))
P I/O
(call,this)

Class-Page

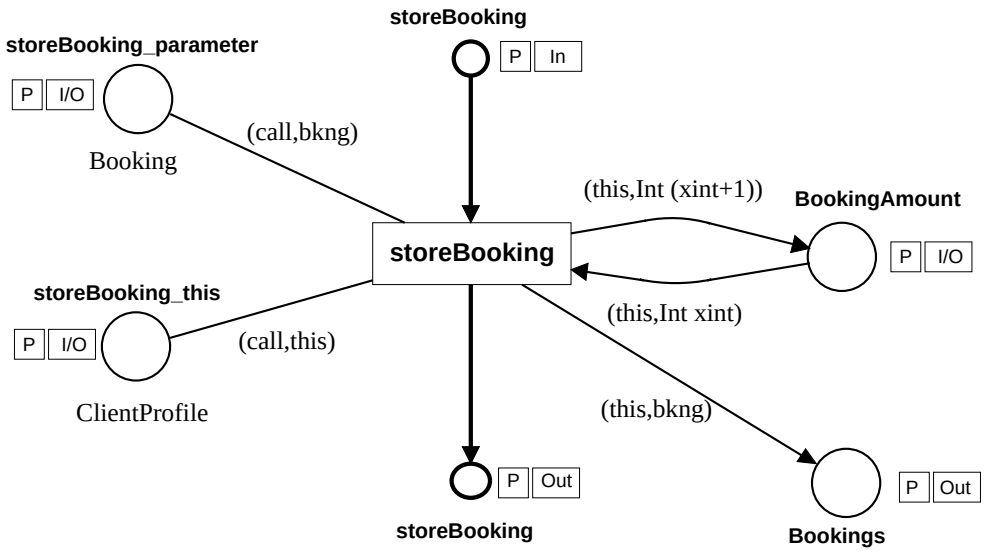

FG
CallMsgP
MSG'
P
I/O
Class
CLASS
ID
1
InstanceCounter
class
oid
oid+1
C class::M"new"::call::msg
(call,I(class,oid)::M"new"::msg)
class
I(class,oid)::M meth::call::msg
(call,I(class,oid)::M meth::msg)
class
[meth <> "new"]
C class::M meth::call::msg
(call,C class::M meth::msg)
P
Out
inMsg
MSG

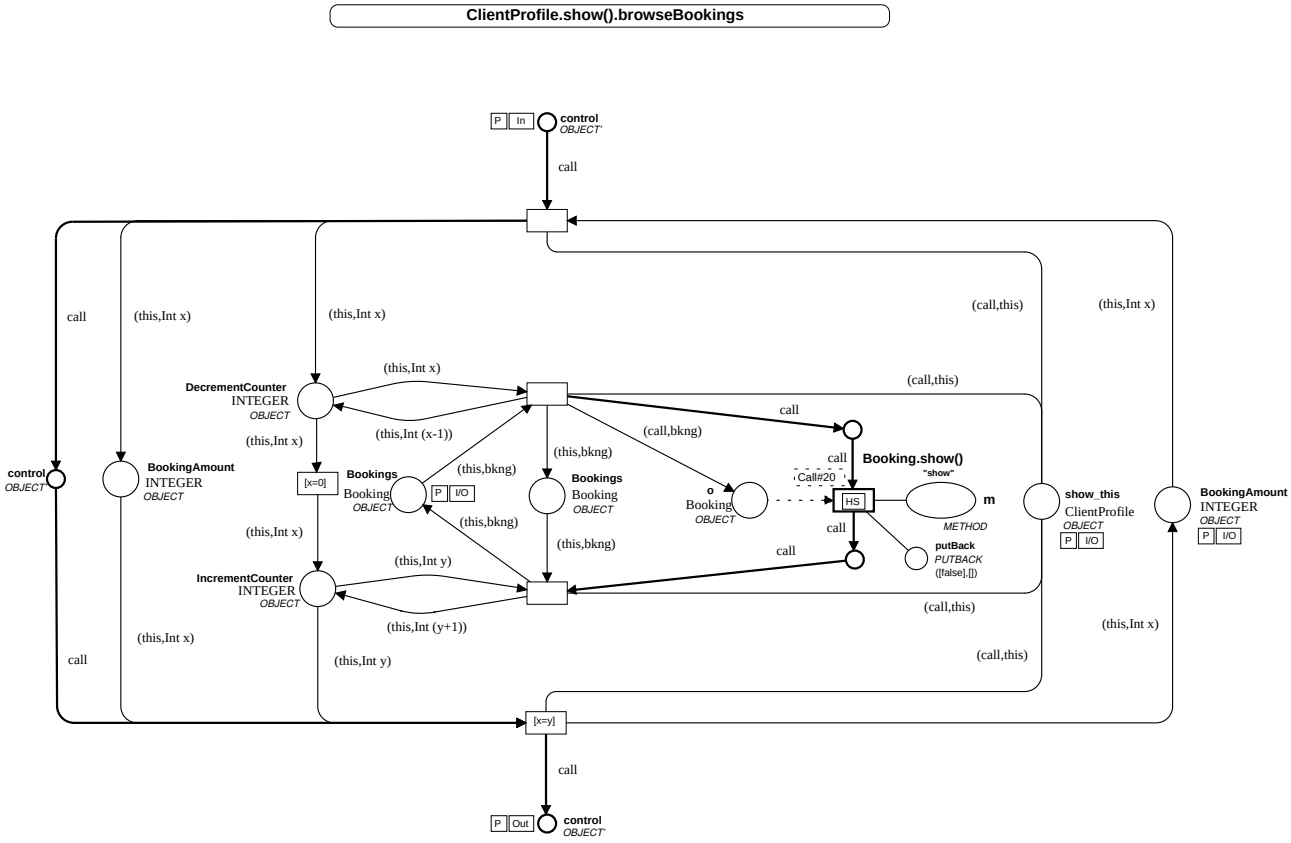
Call-Page

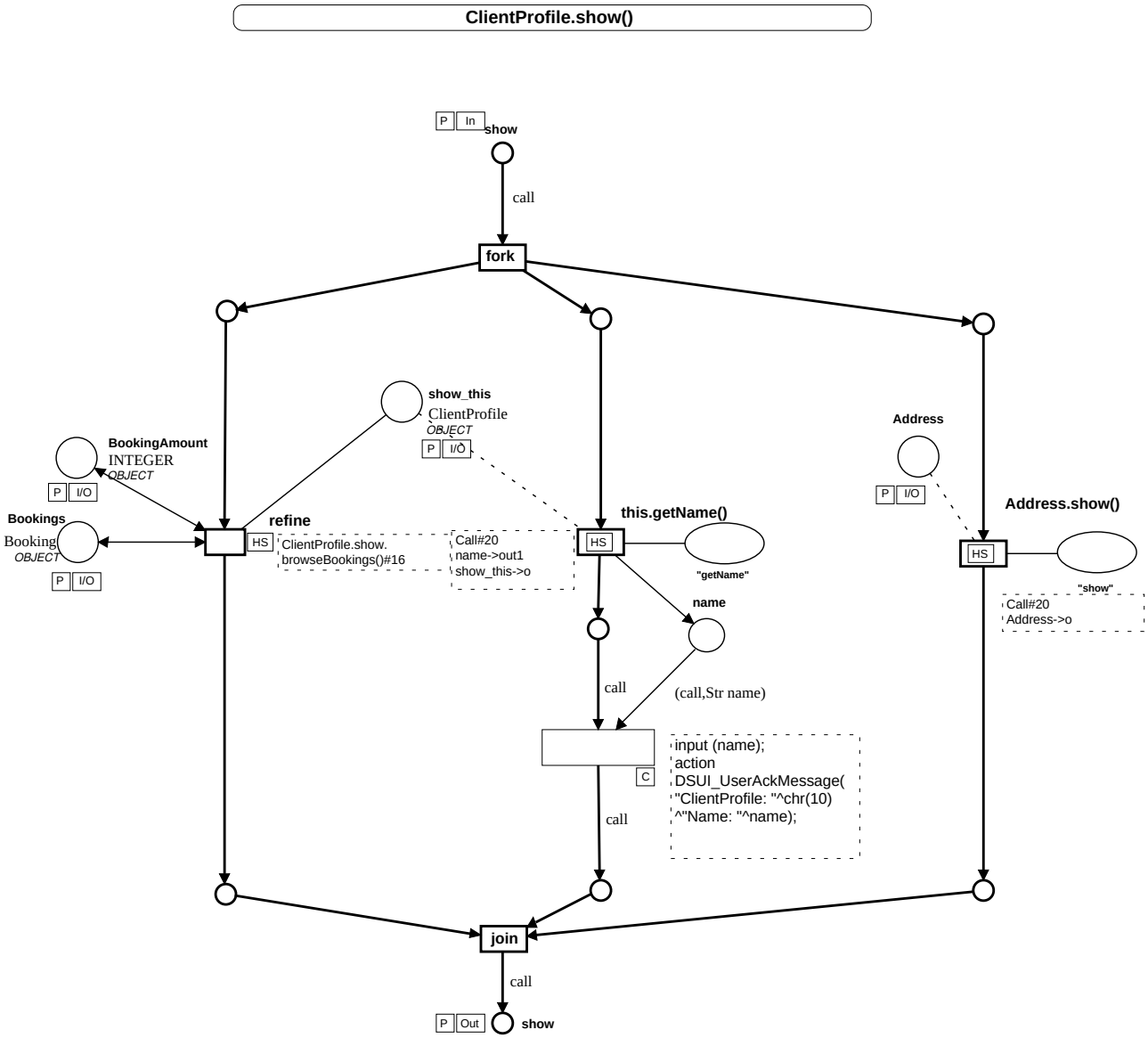



```
input (control,id,this,id1,rcv,id2,meth,id3,in1,in2,cid,in3,id4);
output(call,caller,msg);
action
  let val cntrl = nonThisOrVoid this [control,id,id1,id2,id3,id4];
    val caller = I("Call",cid);
in (cntrl,caller,rcv::M meth::caller::(noVoid [in1,in2,in3]))
end;
```

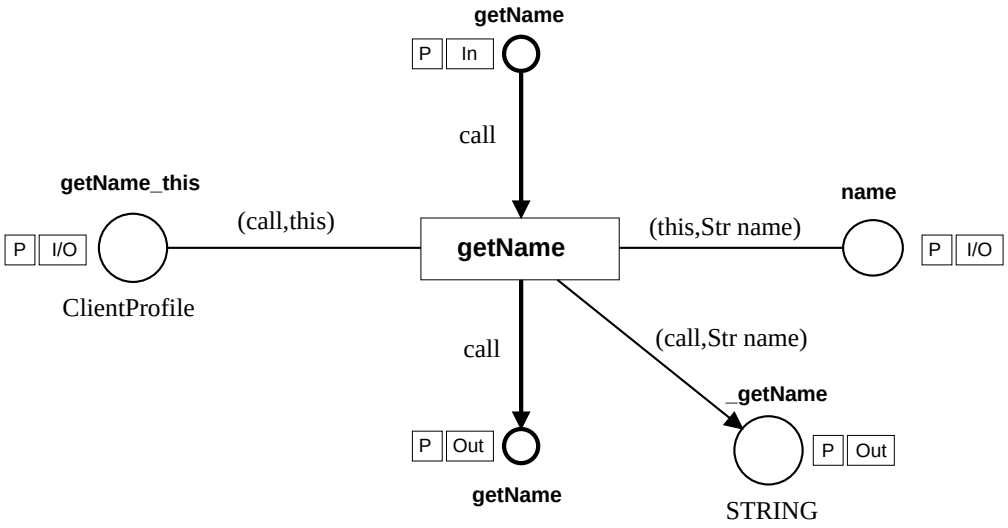
ClientProfile.storeBooking(bkg:Booking)



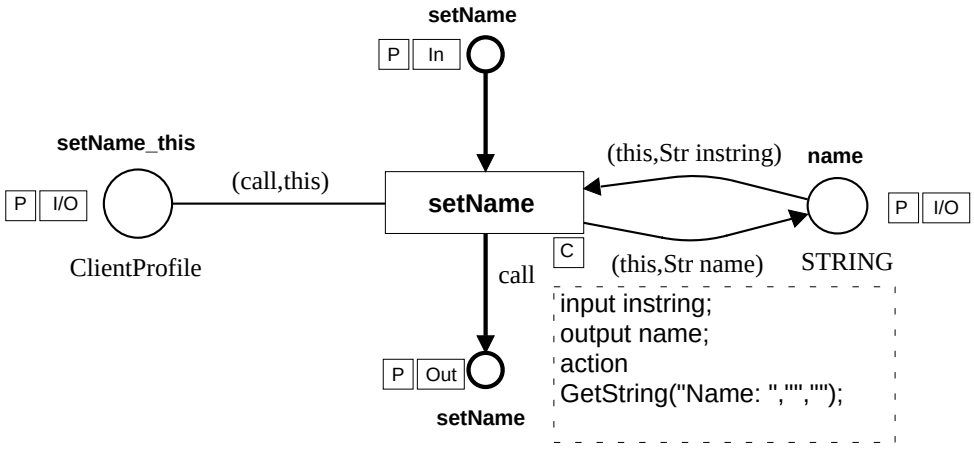

ClientProfile.show().browseBookings
P In
control
OBJECT'
call
(this,Int x)
(call,this)
DecrementCounter
INTEGER
OBJECT
(this,Int (x-1))
(call,bkng)
(this,bkng)
BookingAmount
INTEGER
OBJECT
[x=0]
Bookings
Booking
OBJECT
P I/O
Booking.show()
"show"
Call#20
HS
m
METHOD
o
Booking
OBJECT
putBack
PUTBACK
([false].[])
show_this
ClientProfile
OBJECT
IncrementCounter
INTEGER
OBJECT
(this,Int y)
(this,Int (y+1))
[x=y]
P Out
control
OBJECT'

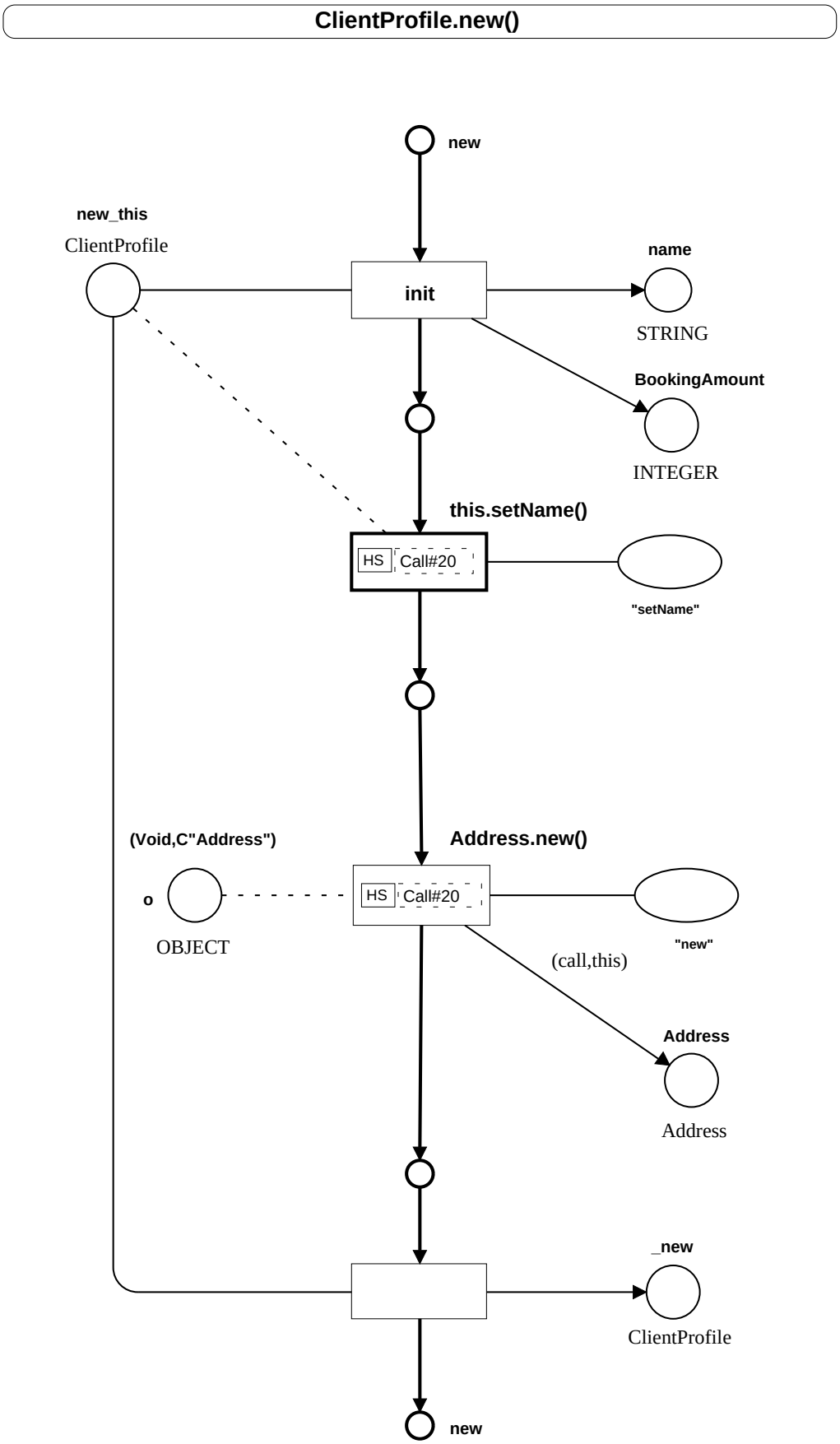

ClientProfile.show()
P In
show
call
fork
show_this
ClientProfile
OBJECT
P I/O
BookingAmount
INTEGER
OBJECT
P I/O
Bookings
Booking
OBJECT
P I/O
refine
HS
ClientProfile.show.
browseBookings()#16
Call#20
name->out1
show_this->o
this.getName()
HS
"getName"
name
call
(call,Str name)
C
input (name);
action
DSUI_UserAckMessage(
"ClientProfile: "^chr(10)
^"Name: "^name);
call
Address
P I/O
Address.show()
HS
"show"
Call#20
Address->o
join
call
P Out
show

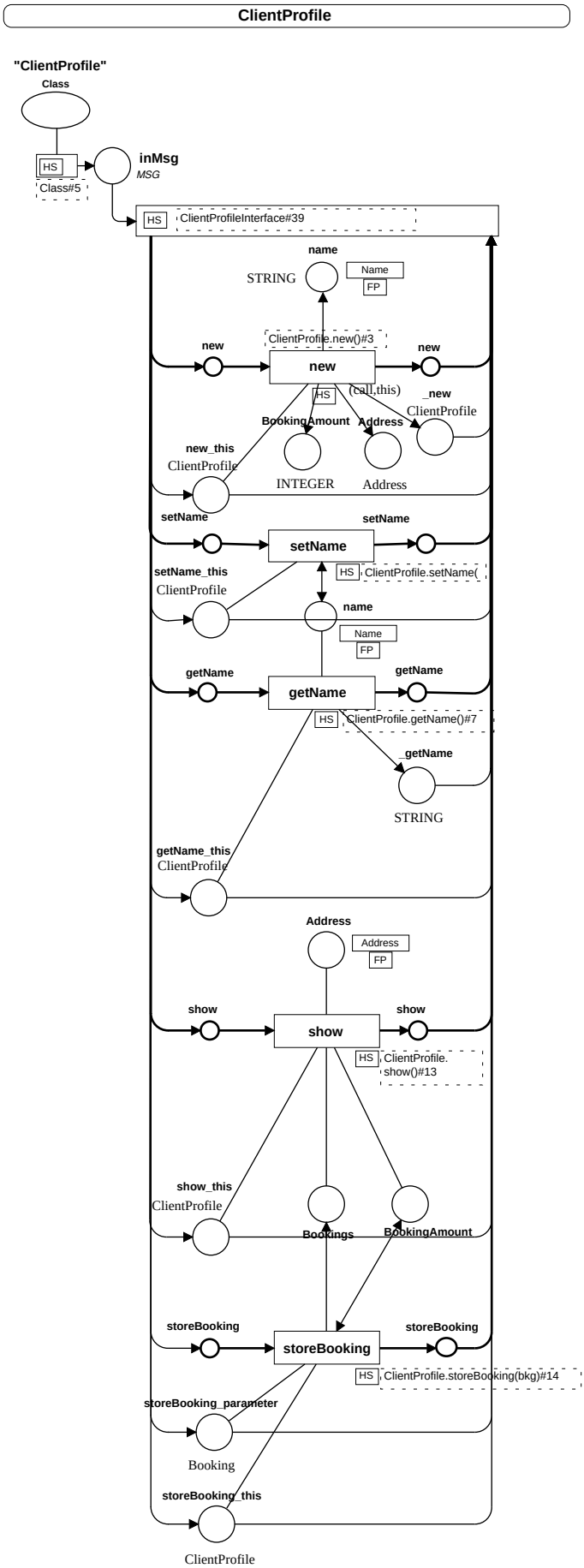
ClientProfile.getName()

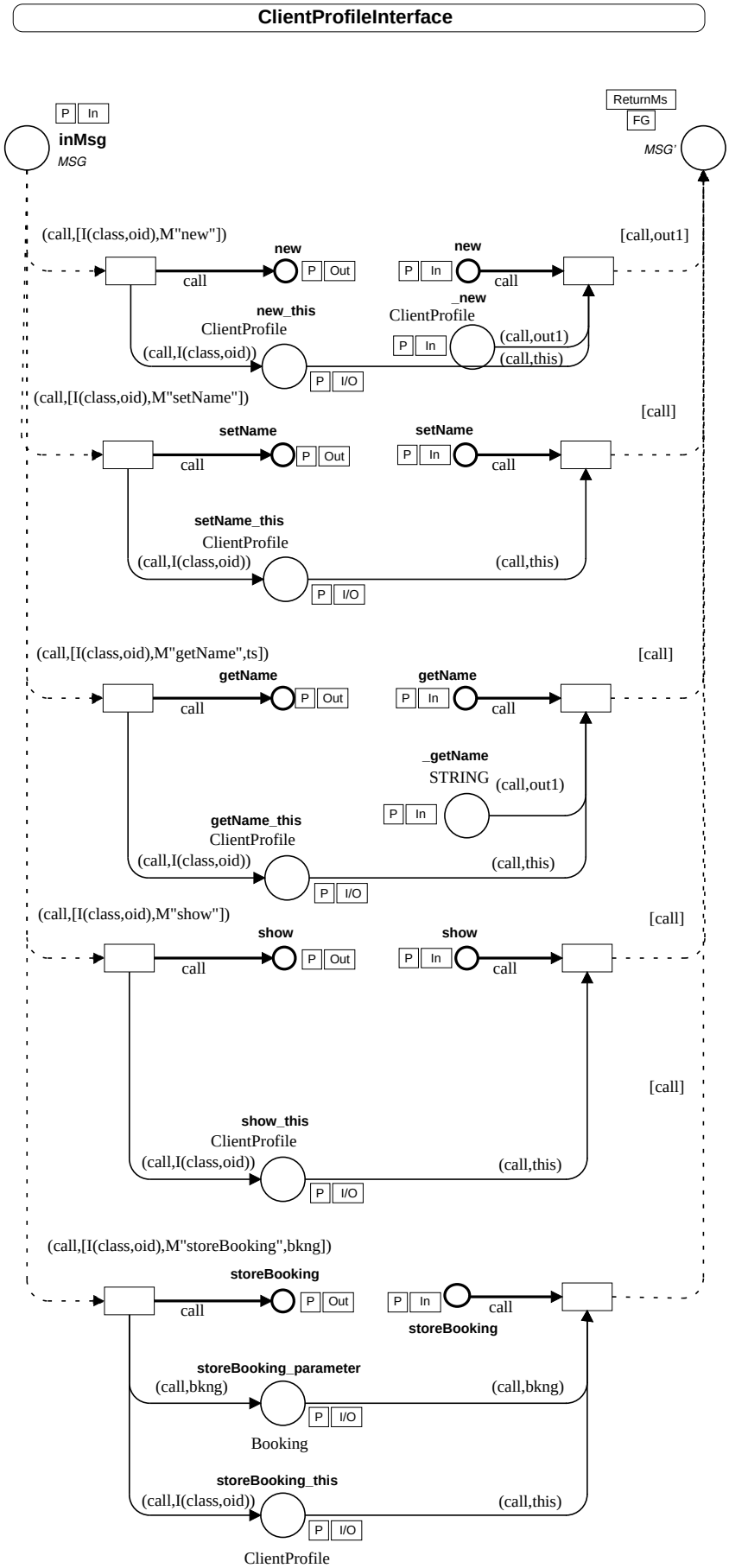


ClientProfile.setName()




ClientProfile.new()
new
new_this
ClientProfile
name
init
STRING
BookingAmount
INTEGER
this.setName()
HS
Call#20
"setName"
(Void,C"Address")
Address.new()
o
HS
Call#20
OBJECT
"new"
(call,this)
Address
Address
_new
ClientProfile
new


ClientProfile
"ClientProfile"
Class
HS
Class#5
inMsg
MSG
HS
ClientProfileInterface#39
name
STRING
Name
FP
new
ClientProfile.new()#3
new
new
HS
(call,this)
_new
ClientProfile
BookingAmount
Address
new_this
ClientProfile
INTEGER
Address
setName
setName
setName
setName_this
ClientProfile
HS
ClientProfile.setName(
name
Name
FP
getName
getName
getName
HS
ClientProfile.getName()#7
getName
STRING
getName_this
ClientProfile
Address
Address
FP
show
show
show
HS
ClientProfile.
show()#13
show_this
ClientProfile
Bookings
BookingAmount
storeBooking
storeBooking
storeBooking
HS
ClientProfile.storeBooking(bkg)#14
storeBooking_parameter
Booking
storeBooking_this
ClientProfile


ClientProfileInterface
P In
inMsg
MSG
ReturnMs
FG
MSG'
(call,[I(class,oid),M"new"])
new
P Out
call
new
P In
call
[call,out1]
new_this
ClientProfile
(call,I(class,oid))
_new
ClientProfile
P In
(call,out1)
(call,this)
P I/O
(call,[I(class,oid),M"setName"])
[call]
setName
P Out
call
setName
P In
call
setName_this
ClientProfile
(call,I(class,oid))
(call,this)
P I/O
(call,[I(class,oid),M"getName",ts])
[call]
getName
P Out
call
getName
P In
call
_getName
STRING
(call,out1)
P In
getName_this
ClientProfile
(call,I(class,oid))
(call,this)
P I/O
(call,[I(class,oid),M"show"])
[call]
show
P Out
call
show
P In
call
[call]
show_this
ClientProfile
(call,I(class,oid))
(call,this)
P I/O
(call,[I(class,oid),M"storeBooking",bkng])
storeBooking
P Out
call
P In
call
storeBooking
storeBooking_parameter
(call,bkng)
(call,bkng)
P I/O
Booking
storeBooking_this
(call,I(class,oid))
(call,this)
P I/O
ClientProfile

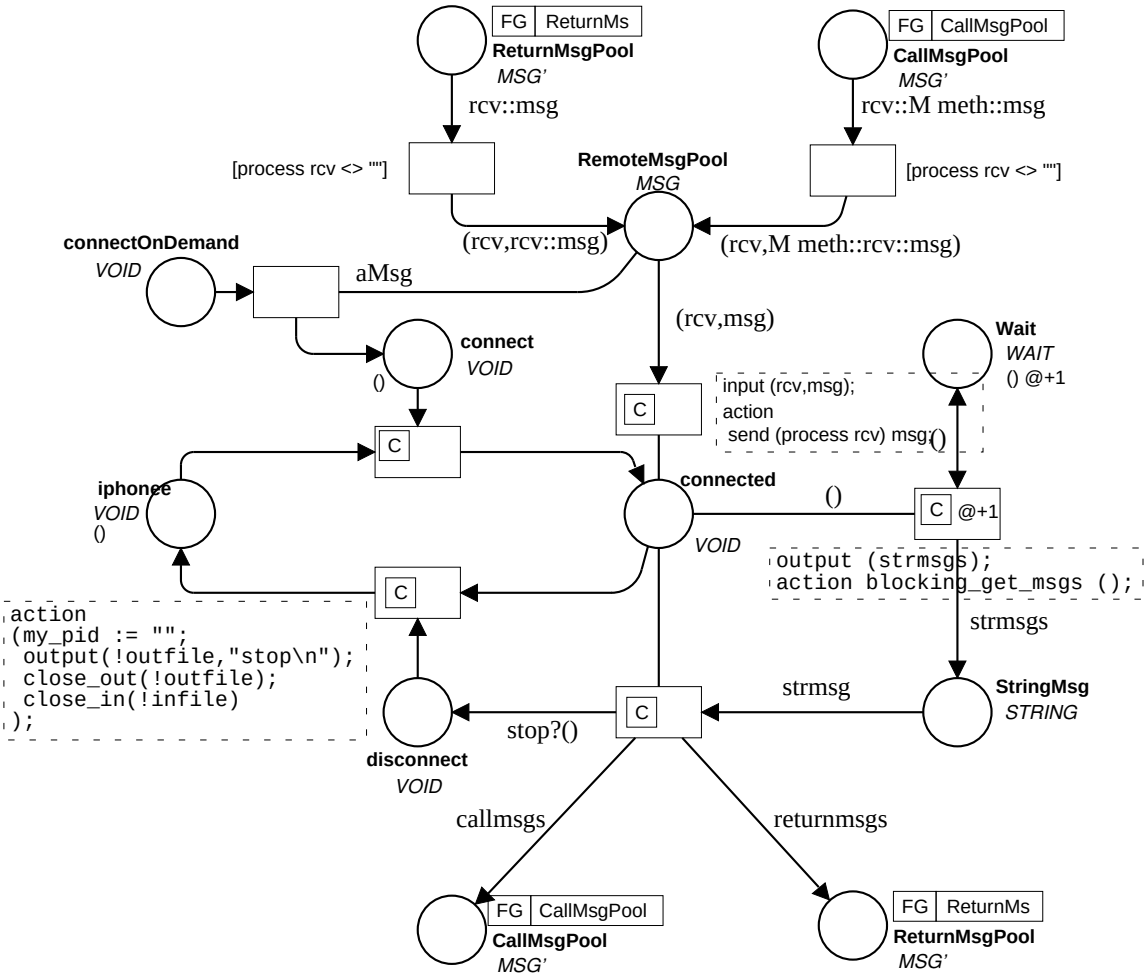
Code-Page

```
(* workaround: *)
val my_execute = execute;

color WAIT = unit timed;
color VOID = unit;
color INTEGER = int;
color REAL = real;
color STRING = string;
color BOOLEAN = bool;
color ID = INTEGER declare input_col;
color RID = STRING;
color BOOLEANLIST = list BOOLEAN;
color INTEGERLIST = list INTEGER;
color PUTBACK = product BOOLEANLIST * INTEGERLIST;
color VALUE = with Integer | RealNumber | String | Boolean;
color MODIFIER = with Static;
color MODIFIERS = list MODIFIER;
color CLASS = STRING;
color METHOD = STRING;
color INSTANCE = product CLASS * ID;
color RINSTANCE = product CLASS * RID * RID;
color RCLASS = product CLASS * RID;
color OBJECT' = union Void + Null
    + C:CLASS + RC:RCLASS
    + M:METHOD
    + I:INSTANCE + RI:RINSTANCE
    + Int:INTEGER + Str:STRING + Bool:BOOLEAN + Real:REAL
    declare mkst_col, of_M;
color MSG' = list OBJECT' declare input_col;
color TYPE = union Object + Reference + ClassRef + InstRef + Value
    + Method + AnyRef + Bottom
    + TC:CLASS + TI:CLASS + TM:METHOD + TV:VALUE;
color TYPELIST = list TYPE;
color MSGTYPE = product TYPELIST * TYPELIST;
color MSGDEF = product MODIFIERS * TYPELIST * METHOD * TYPELIST;
color RETURN' = product OBJECT' * TYPELIST;
color RETURN = product OBJECT'  * RETURN';
color MSG = product OBJECT'  * MSG';
color OBJECTLIST = MSG;
color OBJECT = product OBJECT'  * OBJECT';
color RIDxMSG' = product RID * MSG';

fun GetString(cr_prompt:string,cr_def:string,canc:string)=
let val Stringvalue =
DSUI_GetString{prompt=cr_prompt,def=cr_def}
in
  if Stringvalue <> "" then
      Stringvalue
  else
    GetString(cr_prompt,cr_def,canc)
  end
  handle _ => canc;

fun GetInteger(cr_prompt:string,cr_def:int,canc:int)=
let val Integervalue =
DSUI_GetIntegerValue{prompt=cr_prompt,def=cr_def}
in
  if Integervalue > 0 then
      Integervalue
  else
    GetInteger(cr_prompt,cr_def,canc)
  end
  handle _ => canc;



(* Net Inscription Abbreviations: *)
infix 5 ?;
fun p?v = if p then 1'v else empty;

infix 8 // ;
fun x // y = (x,y);

local fun mkst_list' _ nil = ""
  | mkst_list' mkst [elem] = mkst elem
  | mkst_list' mkst (elem::lst) = mkst elem ^ "," ^ mkst_list' mkst lst
in fun mkst_list mkst lst = "[" ^ mkst_list' mkst lst ^ "]" end;

val msgToStr = mkst_list mkst_col'OBJECT';

fun strToMsg strmsg = (input_col'MSG' (open_string strmsg)) handle _ => [];

local fun remLocal' pid (C(class)) = RC(class,pid)
  | remLocal' pid (I(class,oid)) = RI(class,pid,makestring oid)
  | remLocal' _ old = old
in fun remLocal pid msg = map (remLocal' pid) msg end;

local fun toLocal' pid (old as RC(class,pid')) = if pid=pid' then C(class) else old
  | toLocal' pid (old as RI(class,pid',roid)) =
    if pid=pid' then I(class,input_col'ID (open_string roid)) else old
  | toLocal' _ old = old
in fun toLocal pid msg = map (toLocal' pid) msg end;


fun getType Void = Bottom
  | getType Null = AnyRef
  | getType (C class) = TC class
  | getType (RC(class,_)) = TC class
  | getType (M meth) = TM meth
  | getType (I(class,_)) = TI class
  | getType (RI(class,_,_)) = TI class
  | getType (Int _) = TV Integer
  | getType (Str _) = TV String
  | getType (Bool _) = TV Boolean
  | getType (Real _) = TV RealNumber;

infix 8 <== ;
fun type1 <== type2 =
 if type1 = type2 then true
 else case (type1,type2) of
   (_,          Object  ) => true
   | (Bottom,   _       ) => true
   | (AnyRef, TC _      ) => true
   | (AnyRef, TI _      ) => true
   | (AnyRef, ClassRef  ) => true
   | (AnyRef, InstRef   ) => true
   | (AnyRef, Reference) => true
   | (TC _  , ClassRef  ) => true
   | (TC _  , Reference) => true
   | (TI _  , InstRef   ) => true
   | (TI _  , Reference) => true
   | (TM _  , Method    ) => true
   | (TV _  , Value     ) => true
   | (_     , _         ) => false;

fun ofType typ_obj = getType obj <== typ;
val ofUser = ofType (TI "User");
val ofTail = ofType (TI "tail");
val ofPair = ofType (TI "Pair");


fun typeCheck nil nil = true
  | typeCheck (typ::types) (obj::objs) =
 ofType typ obj andalso typeCheck types objs
  | typeCheck _ _ = false;

fun nonThisOrVoid _ nil = Void (* darf nicht vorkommen! *)
  | nonThisOrVoid this (x::objs) = if x=Void orelse x=this then nonThisOrVoid this objs
          else x;

local
fun sameOrThisOrVoid' y _ nil = y <> Void
  | sameOrThisOrVoid' Void this (x::objs) =
  if x=Void orelse x=this then sameOrThisOrVoid' Void this objs
          else sameOrThisOrVoid' x this objs
  | sameOrThisOrVoid' y this (x::objs) =
  if x=Void orelse x=this orelse x=y then sameOrThisOrVoid' y this objs
          else false
in val sameOrThisOrVoid = sameOrThisOrVoid' Void end;

fun noVoid nil = nil
  | noVoid (Void::xs) = noVoid xs
  | noVoid (x::xs) = x::noVoid xs;

fun prefix xs 0 = nil
  | prefix (x::xs) n = x::prefix xs (n-1)
  | prefix nil _ = nil;

fun put (x::xs) 1 = x
  | put (x::xs) n = put xs (n-1)
  | put nil _ = true;

fun in_n call (obj::objs) 1 = 1'(call,obj)
  | in_n call (_::objs) n = in_n call objs (n-1)
  | in_n call nil _ = empty; (* 1'(call,Null); *)

local fun position' _ nil _ = 0
  | position' n (x::xs) y = if x = y then n else position' (n+1) xs y
in val position = position' 1
end;

fun out_n outs call msg index =
 if mem outs index
  orelse (length outs < -index andalso length msg >= -index)
 then let val pos = if mem outs index then position outs index
          else -index
   in 1'(call,nth(msg,pos-1)) end
 else empty;

fun process (RC(_,pid)) = pid
  | process (RI(_,pid,_)) = pid
  | process _ = "";

globref my_pid = "";
globref infile = std_in;
globref outfile = std_out;

fun send pid msg =
 (output(!outfile, "send\n"^ pid ^"\n" ^
  msgToStr(remLocal (!my_pid) msg) ^"\n");
    flush_out(!outfile));

val callcnt = ref 0;

infix 8 \;
fun (obj\meth) params =
  let val caller = I("CALL",(callcnt:= (!callcnt)+1;!callcnt))
  in (send (process (hd (remLocal (!my_pid) [obj])))
           (M meth::obj::caller::params);
           caller)
  end;

infix 8 return;
fun (obj return params) =
  (send (process (hd (remLocal (!my_pid) [obj])))
        (obj::params));

fun blocking_get_msgs () =
    (1,input_line(!infile)) !!
   (if can_input(!infile)>0
  then blocking_get_msgs ()
  else empty);
```

```
var localmsgs, callmsgs, returnmsgs: MSG' ms;
var strmsg: STRING;
var strmsgs: STRING ms;
var timer, timer': INTEGER;
var pid,roid: RID;
var pbs: BOOLEANLIST;
var outs: INTEGERLIST;
var msg: MSG';
var aMsg: MSG;
var onDemand, stop: BOOLEAN;
var class: CLASS;
var msgType: MSGTYPE;
var inClasses, outClasses: TYPELIST;
var cid,oid: ID;
var control,call,caller,
    id, id', id1, id2, id3, id4: OBJECT';
var meth: METHOD;
var obj, in1,in2,in3,out1,out2: OBJECT';
var this,classobj,snd,rcv,
 lst,pair,cons,tail,add,
    filter,filteredList,
 mapObj,filterObj: OBJECT';
var newHead: OBJECT';
var head,car,cdr,loginstring, parameter, parameter1, parameter2: OBJECT';
var isEmpty, isString, toString, result:OBJECT';
var this'cons,this'cons',
 this'left,this'left',
 this'right,this'right': OBJECT';

var tsc,btc,bc,bus,ts,bt,busID,cp,bkng:OBJECT';
var instring,choicestring,street,town,phone,name,bookno,status,expdate:
STRING;
var x,y,inint,xint,yint,zint,tSeats,aSeats,eSeats,busNo:INTEGER;
var object, choice, loginout:OBJECT;
```

Remote-Page

```
action
 let val name = "ClientProfileProcess";
  val (inf,outf) = my_execute("/home/tgi_3/tgi3/bin/messenger",[name]);
 in (my_pid := name; infile := inf; outfile := outf) end;
```

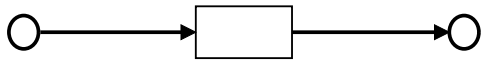



```
input (strmsg);
output (callmsgs,returnmsgs,stop);
action
 case toLocal (!my_pid) (strToMsg (strmsg)) of
    M(m)::rcv::param => (1‘(rcv::M(m)::param), empty, false)
  | I(i)::msg        => (empty,         1‘(I(i)::msg), false)
  | msg              => (empty,         1‘msg,         true);
```

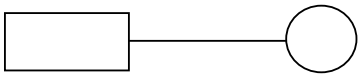
Notationen

Elemente der objektorientierten Petrinetze

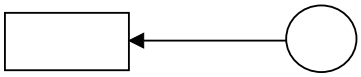
Kontrollfluss



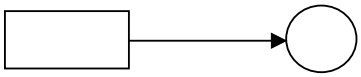
Lesekante



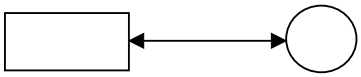
Get-Kante



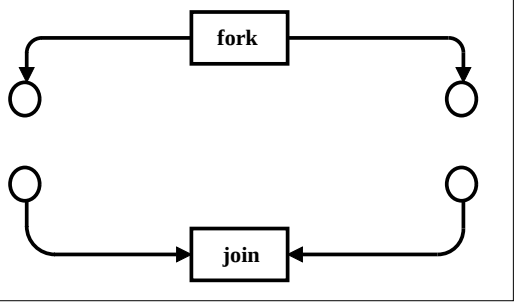
Put-Kante



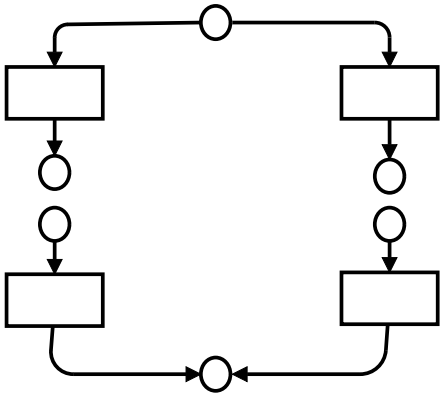
Get-and-Put-Kante



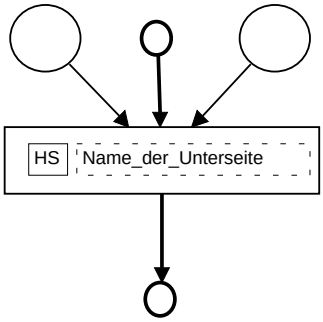
Nebenlaeufiger Kontrollfluss



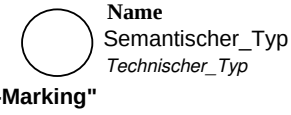
Alternativer Kontrollfluss



Verfeinerung einer Transition auf einer Unterseite

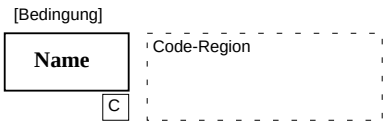


Beschriftung einer Stelle

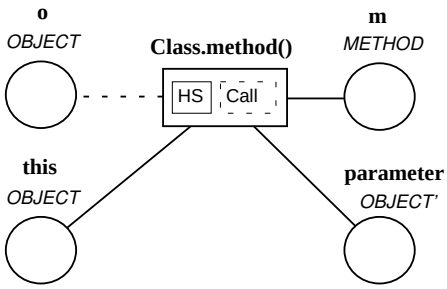


"Initial-Marking"

Beschriftung einer Transition



Verwendung der Call-Unterseite zum Methodenaufruf



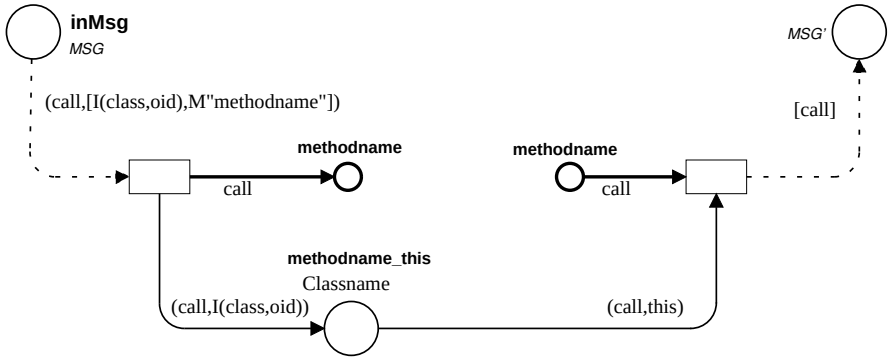
Port-Stellen fuer die Verbindung von Ober- und Unterseiten

Global- und Seiten-Fusions-Stelle

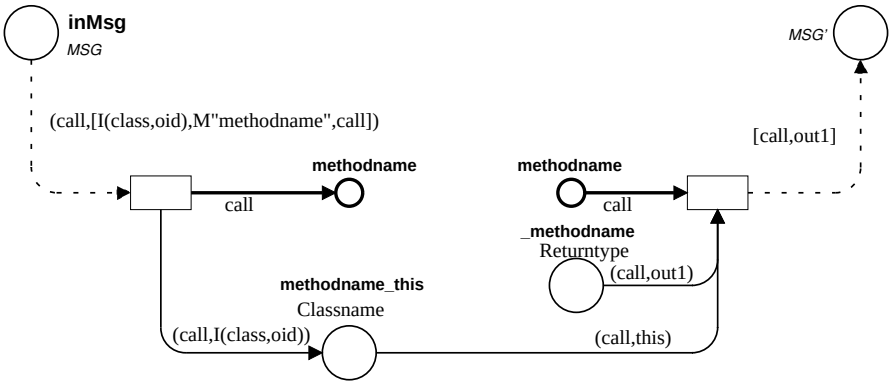
Notation der Interface-Seite

Methodeninterface ohne Parameter und ohne Rueckgabewert

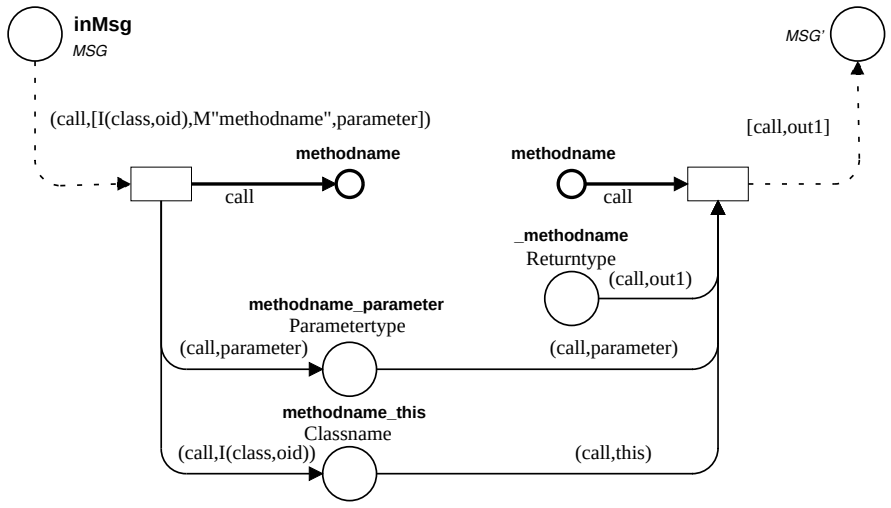
Methodeninterface ohne Parameter, ohne Rueckgabewert und mit Selbst-Referenz



Methodeninterface ohne Parameter, mit einem Rueckgabewert und Selbst-Referenz

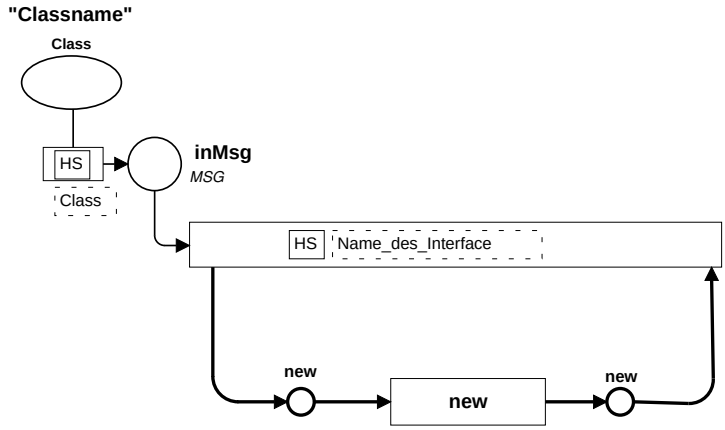


Methodeninterface mit einem Parameter, mit einem Rueckgabewert und Selbst-Referenz

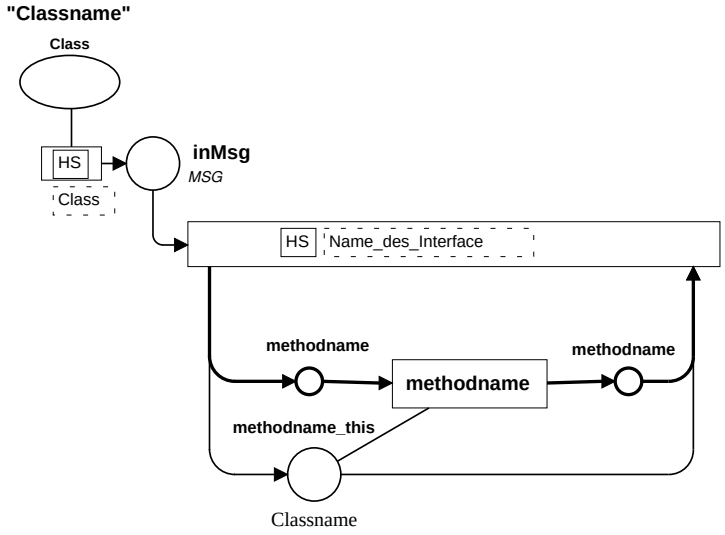


Notation der Klassen-Seite

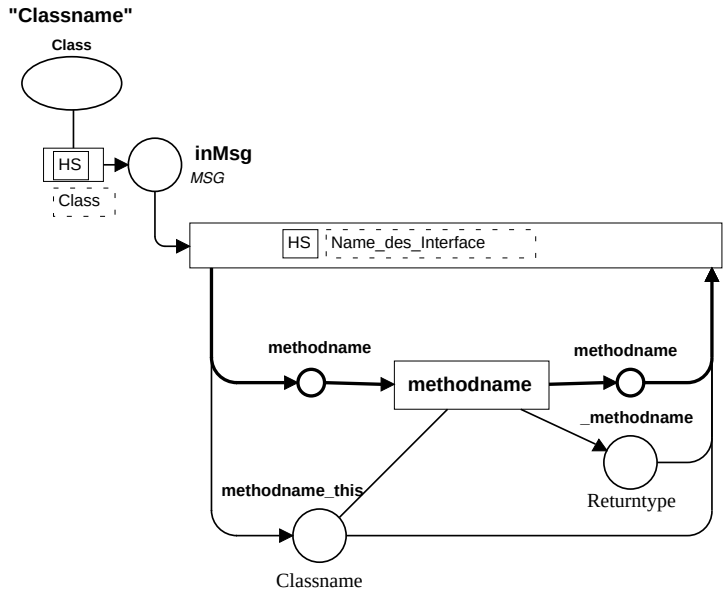
Klasse mit einer new-Methode, ohne Parameter, ohne Selbst-Referenz und ohne Rueckgabewert



Klasse mit einer Methode ohne Parameter, mit Selbst-Referenz und ohne Rueckgabewert

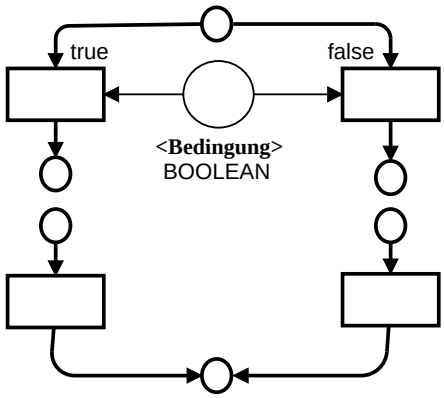


Klasse mit einer Methode ohne Parameter, mit Selbst-Referenz und mit Rueckgabewert

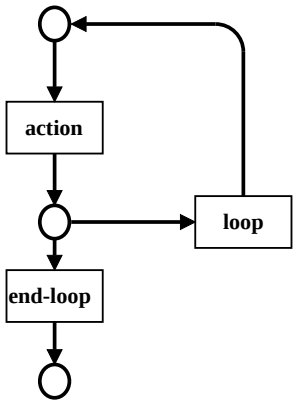


Notation von Kontrollstrukturen I

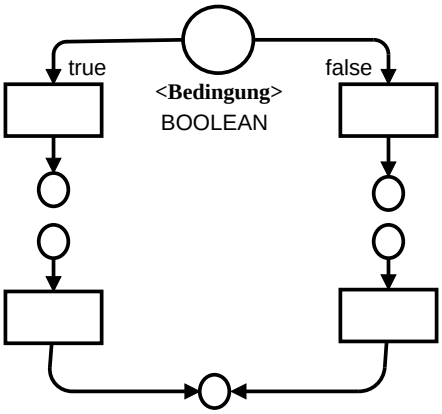
IF-Struktur (Alternative 1)



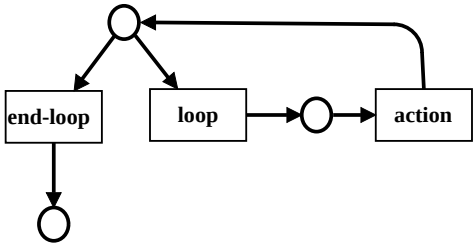
Repeat-Until-Schleife



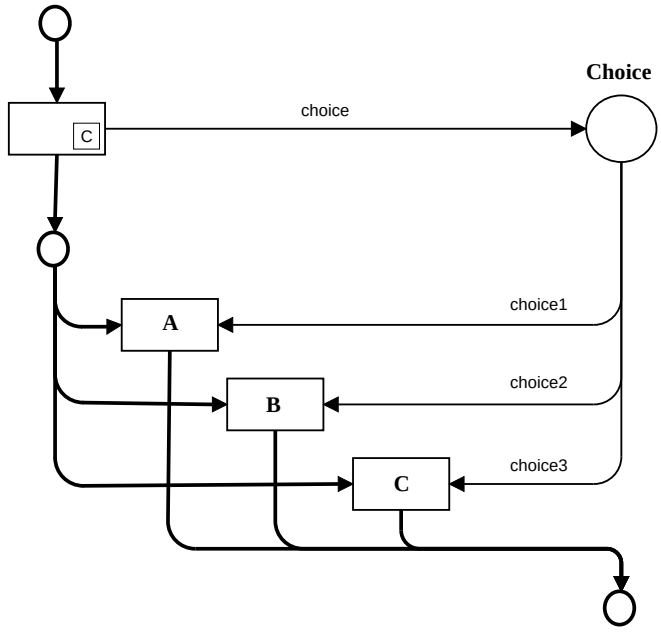
IF-Struktur (Alternative 2)



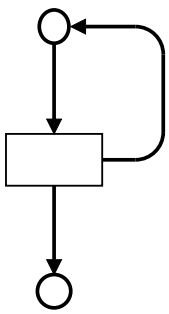
While-Schleife



Case-Struktur

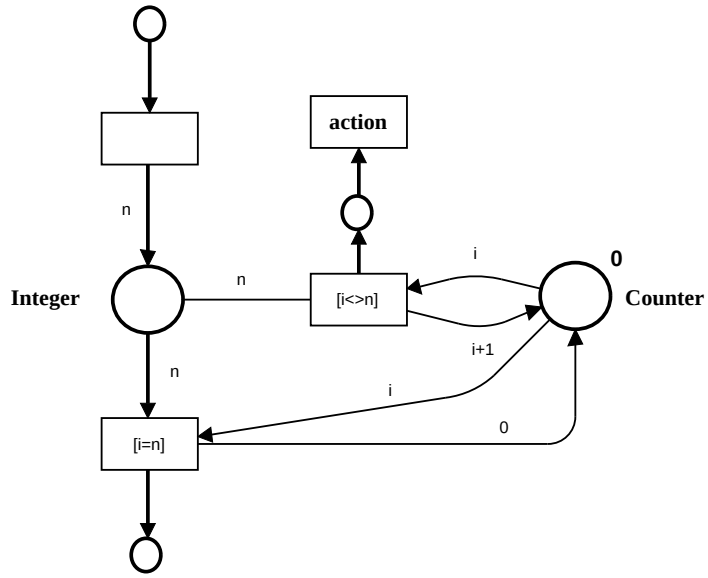
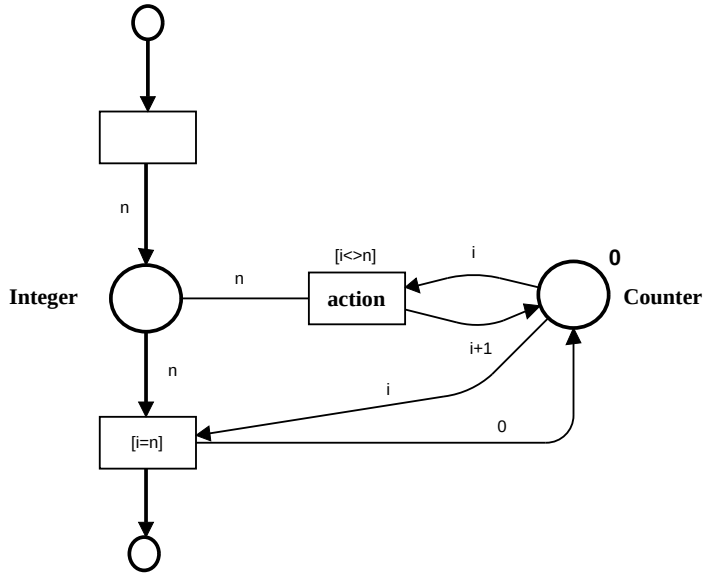


Allgemeine Schleifendarstellung



Notation von Kontrollstrukturen II

Asynchrone FOR-Schleife



Synchrone FOR-Schleife