Sigma Systems, Inc.

Process Diagrams & Descriptions for KSA

Sigma Systems
March 2012

Process Models: KSA-RM

Change Log

Author	Date	Changes
Paul	3/17/2012	Started modeling basic RM processes.
Paul	4/6/2012	Added most of the transaction and account services.
Paul	4/9/2012	Added most of the information services. Rearranged document into specific services.
Paul	5/6/2012	Cleaned up processing for adding transactions to ease UI integration.
Paul	5/21/2012	Added extra transaction processes, specifically to clean up the XML import
Paul	5/21/2012	Completed deferment processes.
Paul	5/23/2012	Memo clarifications, slight alteration to makeEffective() for real-time GL transmissions.
Paul	5/31/2012	Import process for XLIFF. GL preparation diagram. Added canPay() to assist with payment application rules.
Paul	6/1/2012	Altered general ledger process to permit date based recognition strings.
Paul	6/8/2012	Minor clarifications to XML import models.
Paul	6/12/2012	Added processAccounts()
Paul	6/15/2012	Refund processes.
Paul	6/20/2012	Utility services for account and transaction, including blocking services. deapplyAllocation, deapplyLockedAllocation()
Paul	6/21/2012	Flow for the unauthenticated web portal.
Paul	6/25-26/2012	AccessControl classes.
Paul	6/27/2012	Account and Access control work, preference control, check refunds, and account preference system refund systems, payoff refunds.
Paul	6/29/2012	Minor changes to process diagrams per Jen Meyer.
Paul	6/29/2012	Changes to transaction validation logic per meeting at UMDCP, processAccounts()
Paul	7/2/2012	Document review per Jen Meyer. Added getAch() and a number of refund process flows, in particular check and ACH refunds, and the ability to group check and ACH refunds into single payments.
Paul	7/3/2012	Refund reversal, refund ACH, and batch refund ACH.
Paul	7/6/2012 7/10/2012	Payment application services. Updates to XML import to clarify.
Paul	7/16/2012	Cash Limit System
Paul	7/17/2012	Increased checks on createTransaction() to take into account blocking service, etc.
Paul	7/18/2012	Added produceBill(), getFutureBalance()
Paul	7/19/2012	Added writeoff logic methods.
Paul	7/23/2012	Additional methods and name changes per Michael. Added produceReceipt()
Paul	7/24/2012	Added processExternalStatement*() methods.
Paul	8/1/2012	Minor changes to allocation/ deallocation processes.
Paul	8/3/2012	Changed makeEffective() methods to calculate both sides of the general ledger transactions. Completely reworked create/remove AllocationAmount/LockedAllocationAmount methods to create general ledger entries. Added session ability to createGeneralLedgerTransaction,

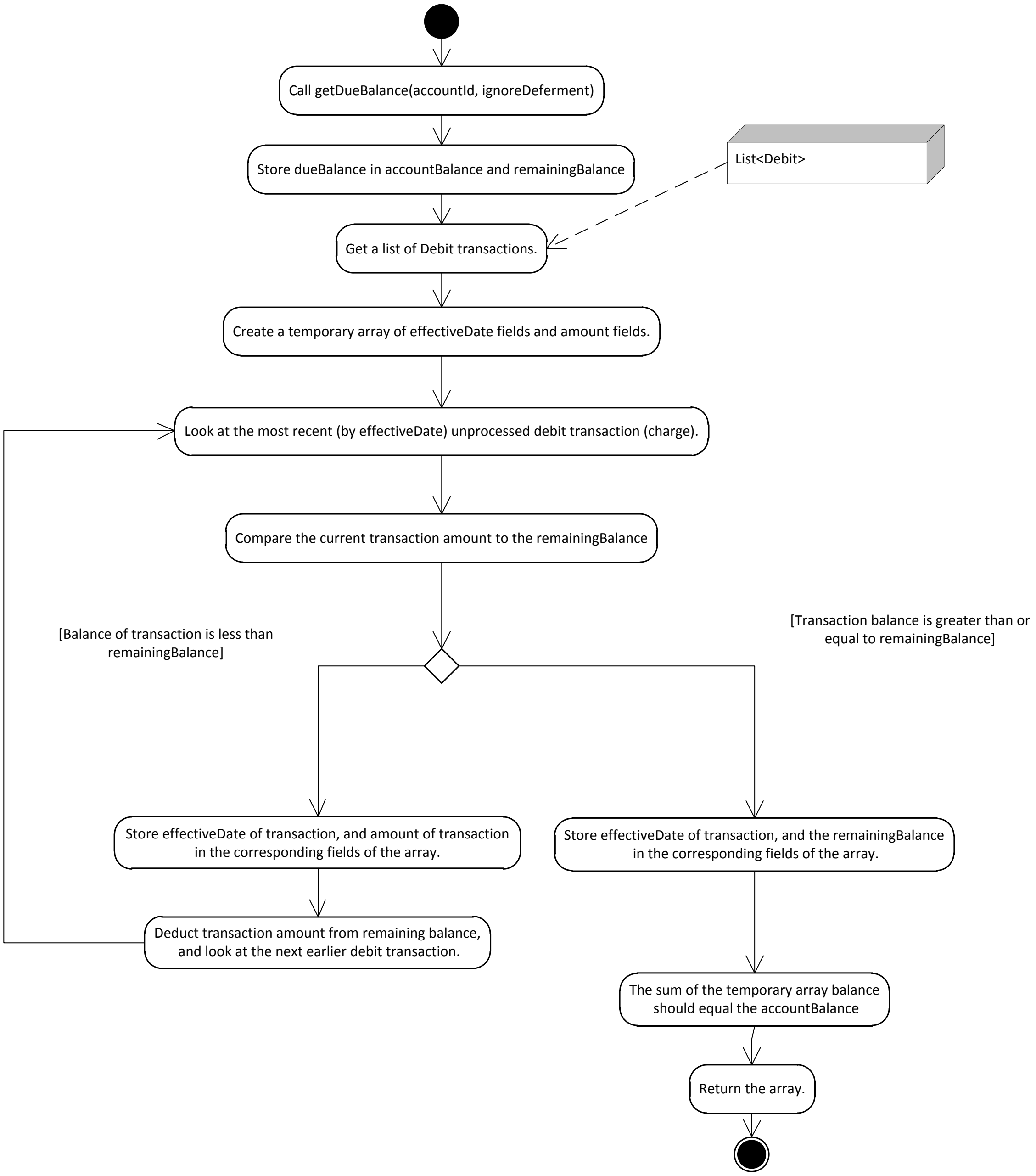
		including taking into account the status H(old).
Paul	8/6/2012	Minor changes to createTransaciton to set general ledger type. Added session cleanup for payment application. Changed expireDeferment methods to permit correct payment application.
Paul	8/7/2012	Minor changes to transaction creation flow from XML due to small changes in XML schema. Added system preference methods.
Paul	8/8/2012	1098T process diagrams.
Paul	8/9/2012	Document review per Jen Meyer. Aged balance reporting. Minor document clean up.
Paul	8/10/2012	Added produceAccountReport()
Paul	8/13/2012	Changes to createAllocation() createLockedAllocation(), removeAllocation(), removeLockedAllocation(), removeAllAllocations(), cleanupSession() (became summarizeGeneralLedgerTransactions(), applyTransactions() and addGeneralLedgerTransactions() to remove the session idea, and allow the passing of the isQueued parameter to improve encapsulation of payment application logic. Removed getUuid() method. Added prepareGeneralLedgerReport()
Paul	8/14/2012	Minor change to createTransaction() to take account of recognitionDate. Minor change to saveTransaction() to call checkCashLimit() routine if needed. Changes to all general ledger methods with recognition periods to take into account the recognitionDate attribute. Added recognition period to produceTransactions()
Paul	8/15/2012	1098T, completed the "next quarter payment" box. Added new payment application methods to simplify tasks for rules. Removed deprecated payment application methods. Corrected spelling mistake in createTransaction(). Minor changes from getBaseTransactionType() to getTransactionTypeClass()

Contents

Account Service

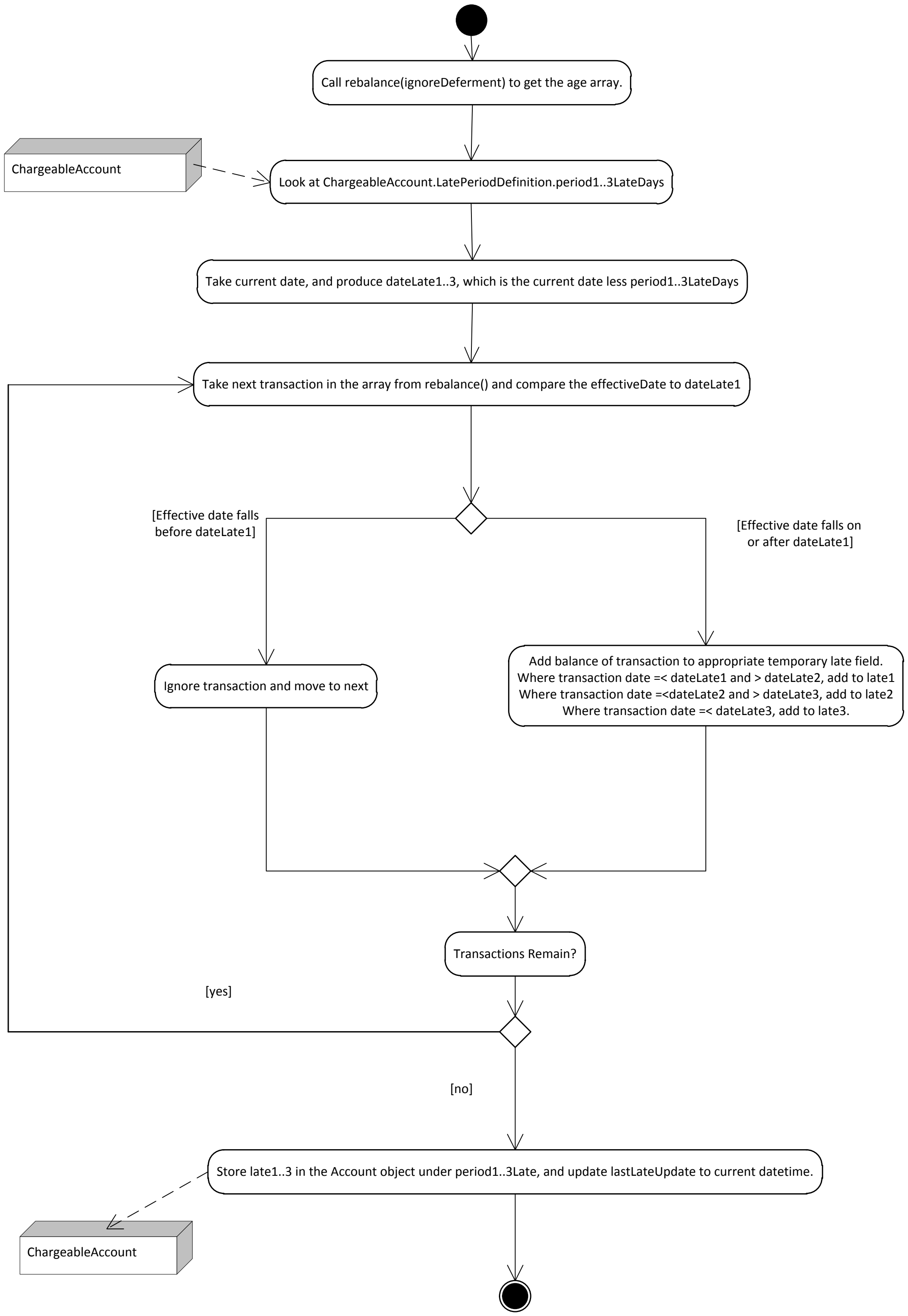
rebalance(accountId, ignoreDeferment)

This process creates a temporary subset of the account as if the account were being administered as a balance forward account. This permits aging the account in a way that is not affected by the payment application methodology. This temporary array is passed to the ageDebt() method.


Call getDueBalance(accountId, ignoreDeferment)
Store dueBalance in accountBalance and remainingBalance
List<Debit>
Get a list of Debit transactions.
Create a temporary array of effectiveDate fields and amount fields.
Look at the most recent (by effectiveDate) unprocessed debit transaction (charge).
Compare the current transaction amount to the remainingBalance
[Balance of transaction is less than remainingBalance]
[Transaction balance is greater than or equal to remainingBalance]
Store effectiveDate of transaction, and amount of transaction in the corresponding fields of the array.
Store effectiveDate of transaction, and the remainingBalance in the corresponding fields of the array.
Deduct transaction amount from remaining balance, and look at the next earlier debit transaction.
The sum of the temporary array balance should equal the accountBalance
Return the array.

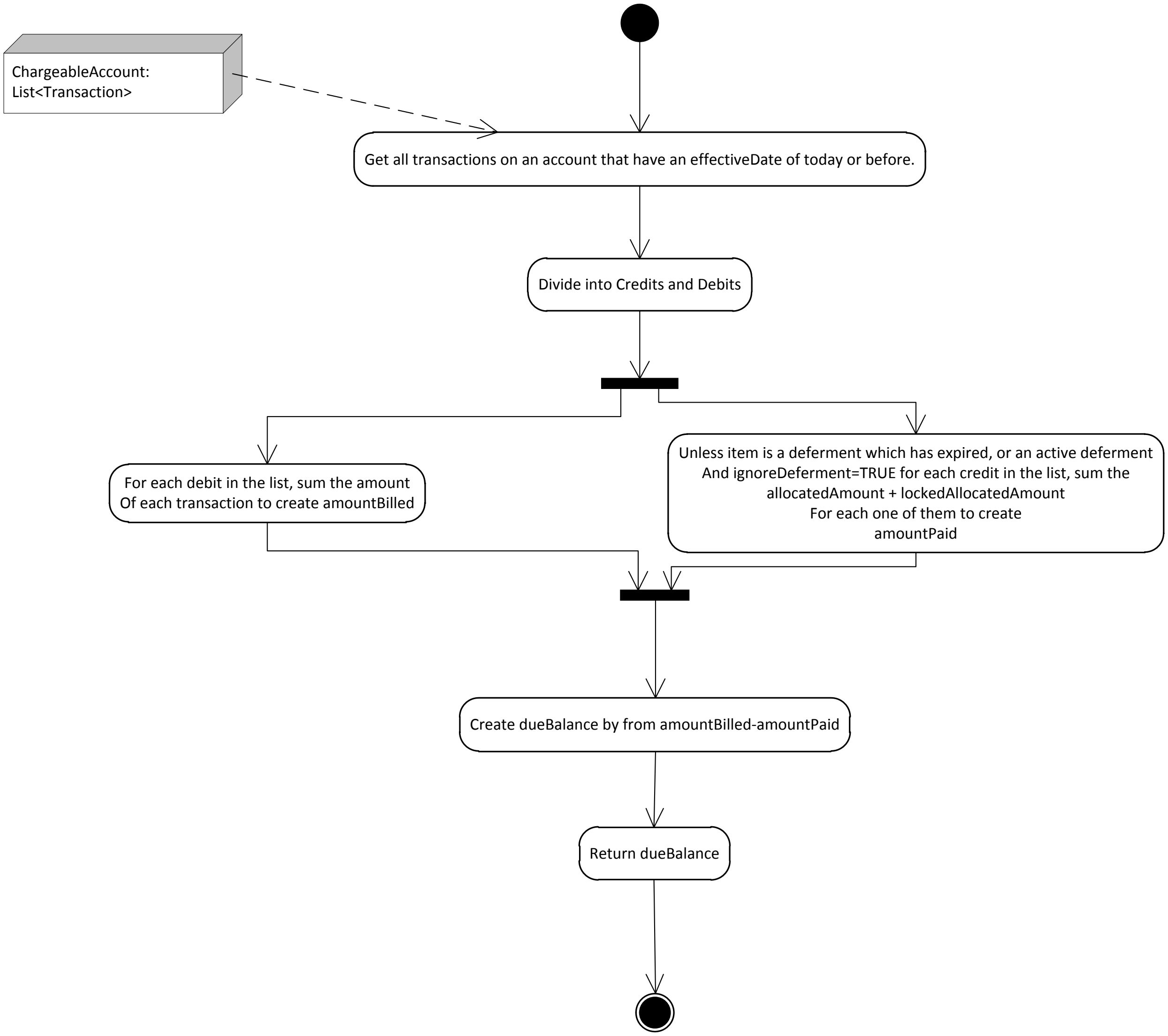
ageDebt(accountId, ignoreDeferment)

ageDebt(list<accountId>, ignoreDeferment)


Call rebalance(ignoreDeferment) to get the age array.
ChargeableAccount
Look at ChargeableAccount.LatePeriodDefinition.period1..3LateDays
Take current date, and produce dateLate1..3, which is the current date less period1..3LateDays
Take next transaction in the array from rebalance() and compare the effectiveDate to dateLate1
[Effective date falls before dateLate1]
[Effective date falls on or after dateLate1]
Ignore transaction and move to next
Add balance of transaction to appropriate temporary late field.
Where transaction date =< dateLate1 and > dateLate2, add to late1
Where transaction date =<dateLate2 and > dateLate3, add to late2
Where transaction date =< dateLate3, add to late3.
Transactions Remain?
[yes]
[no]
Store late1..3 in the Account object under period1..3Late, and update lastLateUpdate to current datetime.
ChargeableAccount

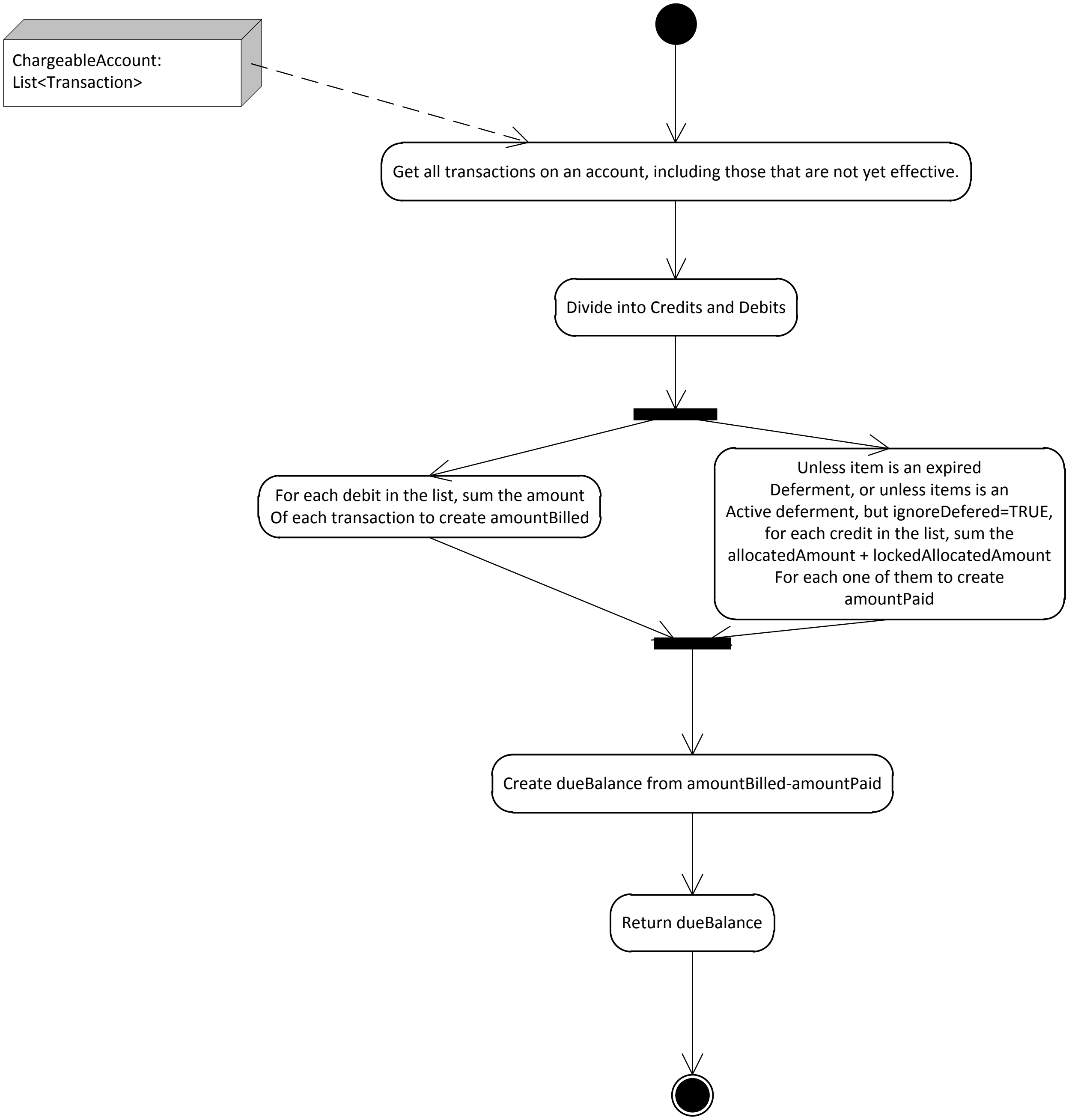
getDueBalance(accountId, ignoreDeferment)

This will get the balance of the account, taking into account only those transactions that are current (effectiveDate is today or before.)

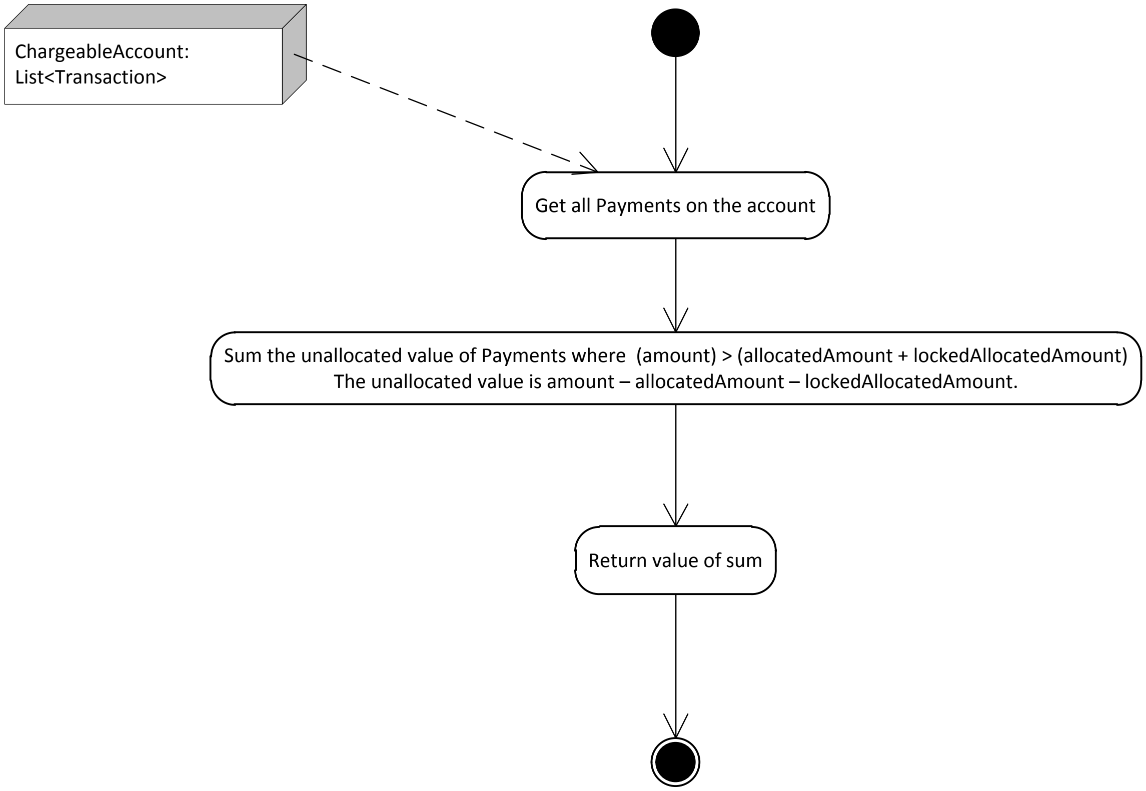

ChargeableAccount:
List<Transaction>
Get all transactions on an account that have an effectiveDate of today or before.
Divide into Credits and Debits
For each debit in the list, sum the amount
Of each transaction to create amountBilled
Unless item is a deferment which has expired, or an active deferment
And ignoreDeferment=TRUE for each credit in the list, sum the
allocatedAmount + lockedAllocatedAmount
For each one of them to create
amountPaid
Create dueBalance by from amountBilled-amountPaid
Return dueBalance

getOutstandingBalance(accountId, ignoreDeferment)

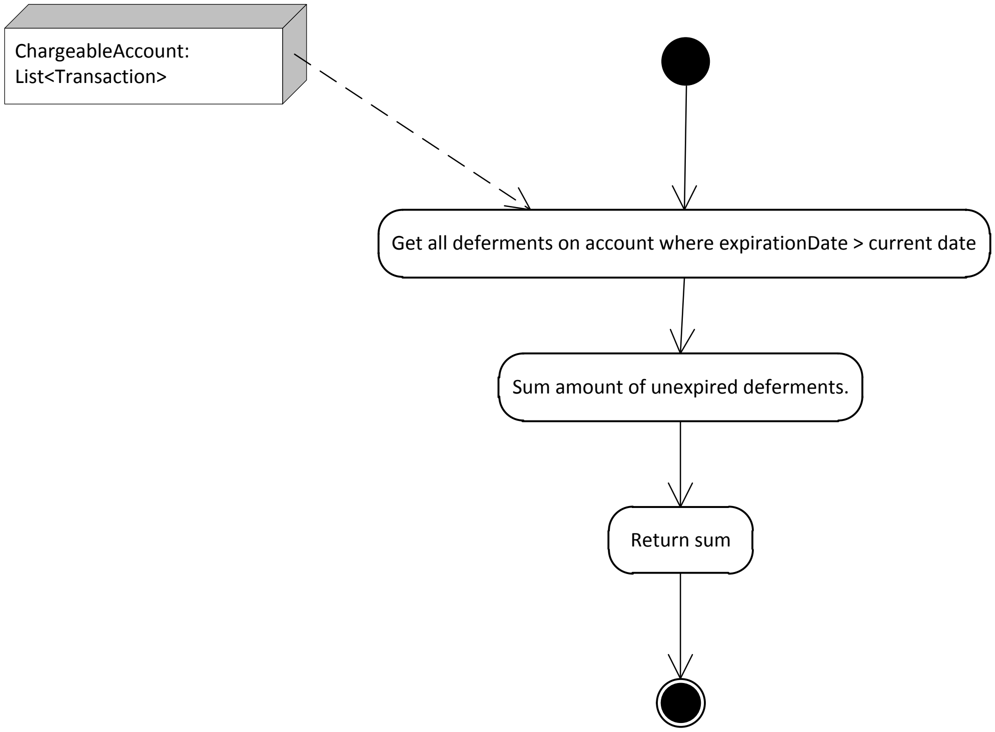
This will get the balance of the account including future dated transactions.


ChargeableAccount:
List<Transaction>
Get all transactions on an account, including those that are not yet effective.
Divide into Credits and Debits
For each debit in the list, sum the amount
Of each transaction to create amountBilled
Unless item is an expired
Deferment, or unless items is an
Active deferment, but ignoreDefered=TRUE,
for each credit in the list, sum the
allocatedAmount + lockedAllocatedAmount
For each one of them to create
amountPaid
Create dueBalance from amountBilled-amountPaid
Return dueBalance

getUnallocatedBalance(accountId)


ChargeableAccount:
List<Transaction>
Get all Payments on the account
Sum the unallocated value of Payments where (amount) > (allocatedAmount + lockedAllocatedAmount)
The unallocated value is amount – allocatedAmount – lockedAllocatedAmount.
Return value of sum

getDeferredAmount(accountId)


ChargeableAccount:
List<Transaction>
Get all deferments on account where expirationDate > current date
Sum amount of unexpired deferments.
Return sum

getFutureBalance (accountId, ignoreDeferment)

do getDueBalance() and getOutstandingBalance() with the same parameters as this method, and then return outstanding – due.

clearAllocations(accountId)

This method is used to clear all non-locked allocations on an account, and permits payment application to start again.

Get all transactions on the account.

Set allocatedAmount to 0 for each transaction.

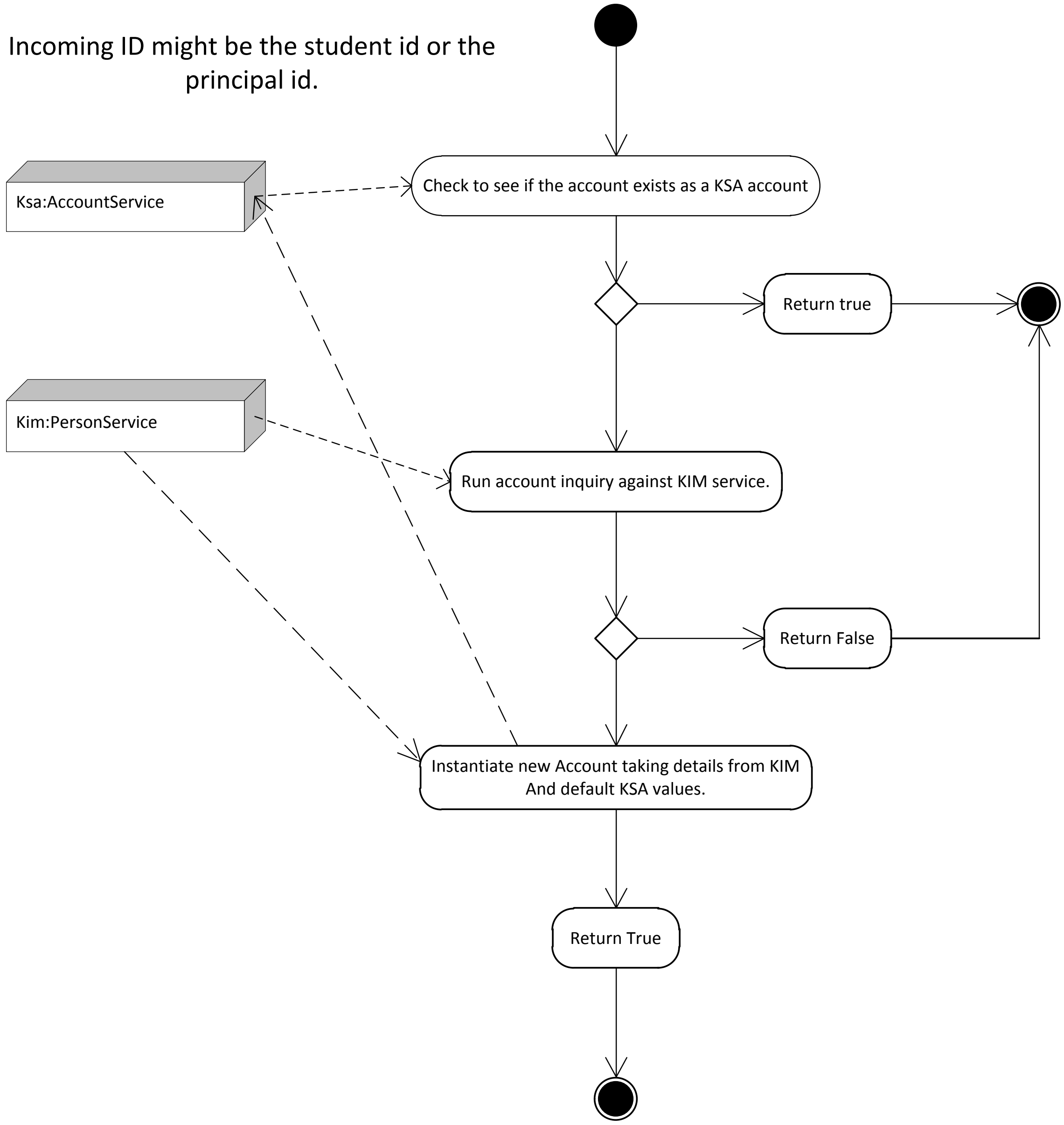
For each transaction, get the Allocation object with the transactionId in either Allocation.firstTransaction or Allocation.secondTransaction, and if isLocked is false, then destroy that object.

clearAllocations (list<accountId>)

For each accountId, call clearAllocations()

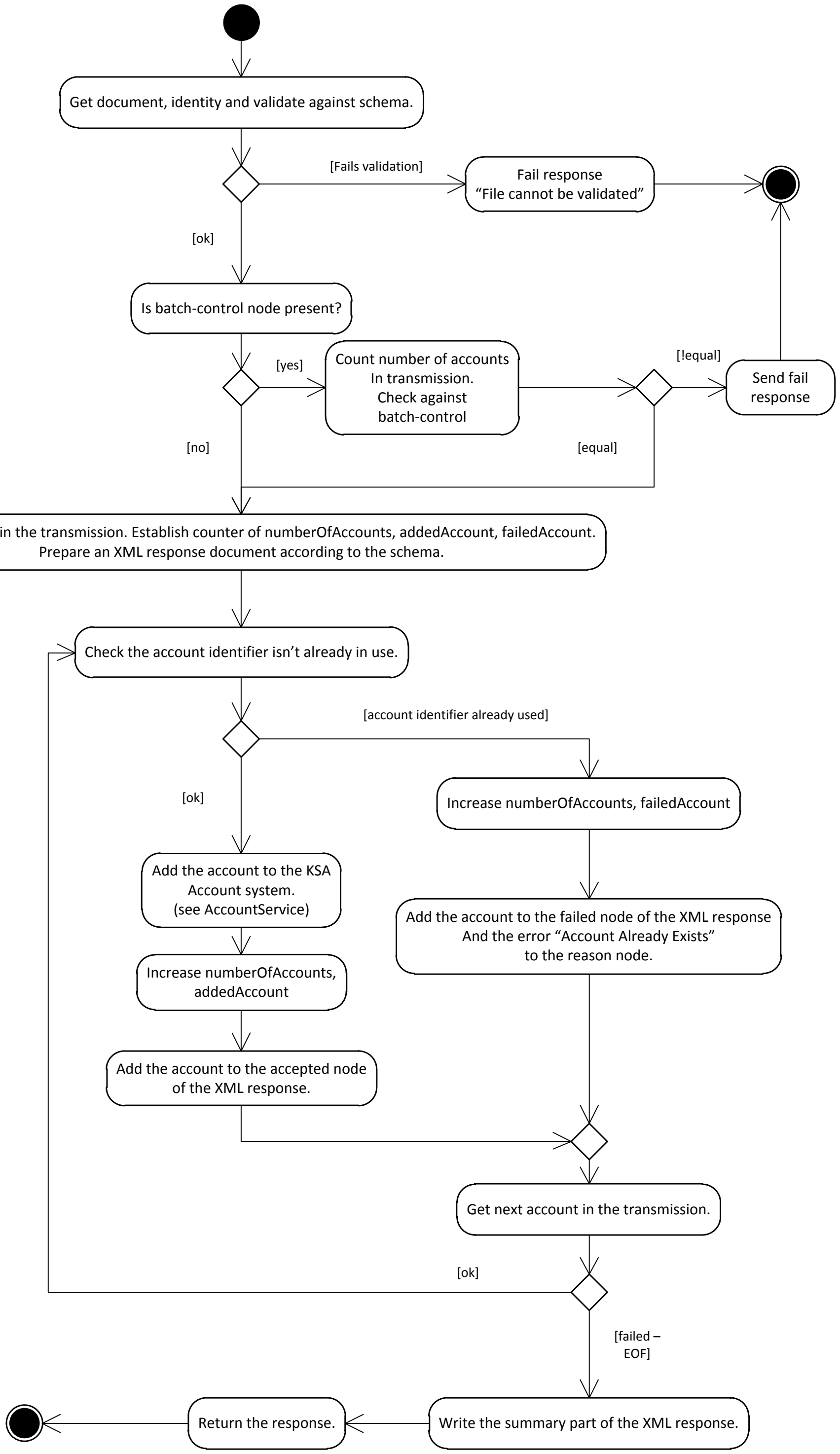
doesAccountExist(accountIdentifer)

This method is used to verify that an account exists before a transaction or other operations are performed on the account. There is an initial inquiry into the KSA store. If no account exists, then there is an inquiry into KIM. If KIM also returns no result, then false is returned. If a KIM account does exist, then a KSA account is created, using the KIM information as a template.


Incoming ID might be the student id or the principal id.
Ksa:AccountService
Check to see if the account exists as a KSA account
Return true
Kim:PersonService
Run account inquiry against KIM service.
Return False
Instantiate new Account taking details from KIM And default KSA values.
Return True

processAccounts (inputFile)

Simple import of accounts via XML.


Get document, identity and validate against schema.
[Fails validation]
Fail response
“File cannot be validated”
[ok]
Is batch-control node present?
[yes]
Count number of accounts
In transmission.
Check against
batch-control
[!equal]
Send fail
response
[no]
[equal]
Get first account in the transmission. Establish counter of numberOfAccounts, addedAccount, failedAccount.
Prepare an XML response document according to the schema.
Check the account identifier isn’t already in use.
[account identifier already used]
Increase numberOfAccounts, failedAccount
[ok]
Add the account to the KSA
Account system.
(see AccountService)
Add the account to the failed node of the XML response
And the error “Account Already Exists”
to the reason node.
Increase numberOfAccounts,
addedAccount
Add the account to the accepted node
of the XML response.
Get next account in the transmission.
[ok]
[failed –
EOF]
Return the response.
Write the summary part of the XML response.

doesKsaAccountExist(accountIdentifier)

Check ONLY if the account exists in the KSA system. Return either true or false.

instantiateKsaAccount()

instantiateKsaAccount(accountIdentifier)

To start building an account, we first need an identifier. There are three types of account number. The first is an account number derived from KIM. In most implementations, this will be the standard student identifier used at the institution. The second is an external system identifier, where an account is made on behalf of another system. This exists to cover the use case of creating an account for an unknown identity, for example, a parking ticket where nothing is known about the account holder other than the identifier for the car. The third is a KSA account number. This is included to cover unknown use cases at this point, but is included for future use. KSA account numbers are prefixed with KSA to reduce possibility of conflicts. The format of KIM account numbers is defined by the institution, and the format of external account numbers is decided by the external system. All KSA will do is ensure there is no conflict between the numbers.

If an id is passed, check to see if the account already exists (doesKsaAccountExist(id)) If there is a KIM account with this id, the account will be generated in that method. If not, then an account framework will be generated, using the id passed. If no id is passed, an account framework is created with the next KSA account number (KSA0001910, etc.) The methods return the id of the account.

Create:

AccountProtectedInfo with only the new account id.

Account: accountId, status=*defaultAccountStatusType*, creationDate=current system datetime. creditLimit=*defaultCreditLimit.*

This gives the system a shell account, into which account details can now be placed.

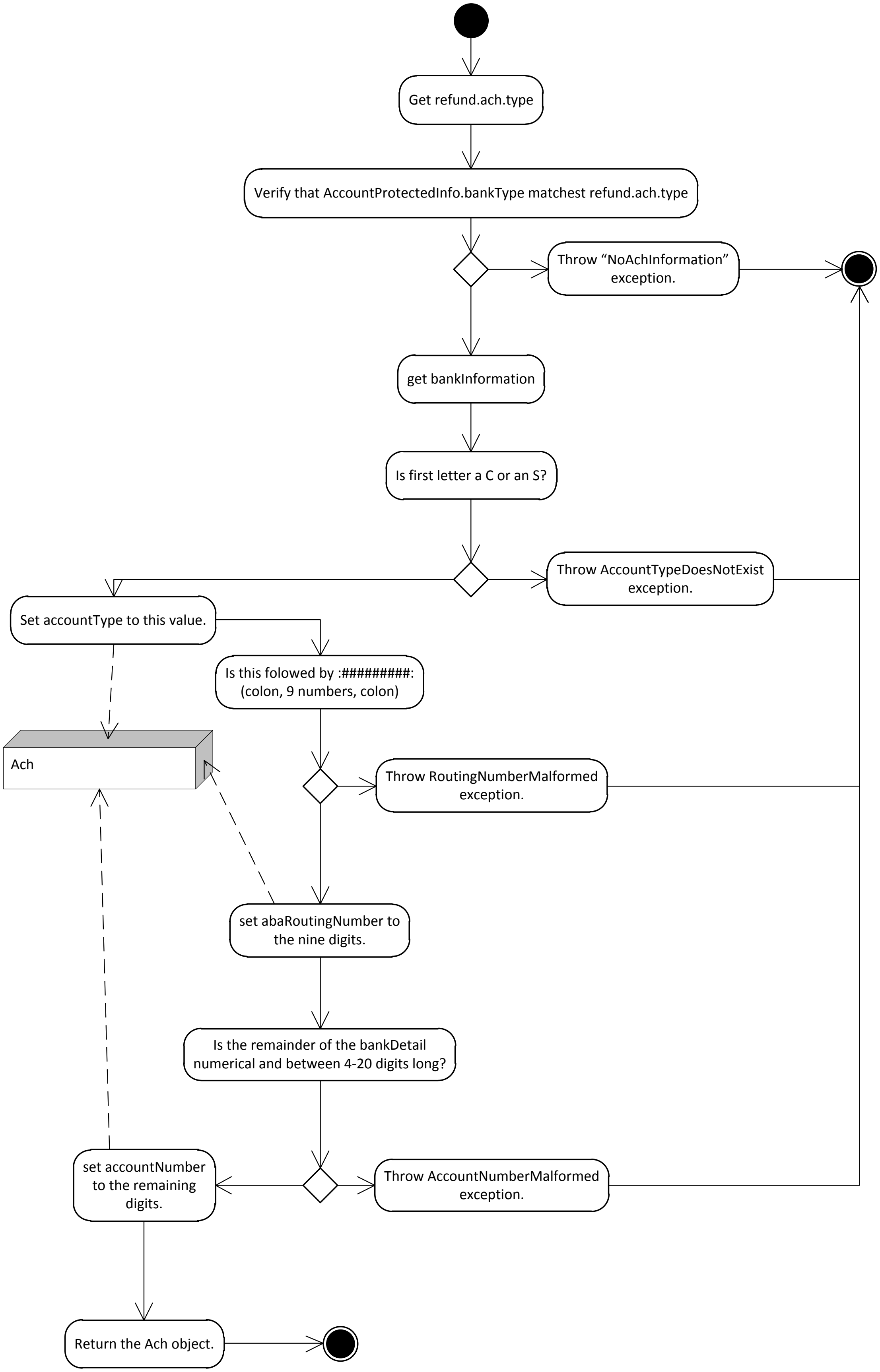
getAccountBlockStatus(accountId)

This method calls a rule-based process to look for any blocks that may apply to the account. An account object is passed to the rules, and any blocks that exist on the account are returned by the method as a group of AccountBlock objects. Note that this method does nothing to enforce the blocks, merely reports which blocks may be in effect on the account. It is the job of other rules to decide what to do with those blocks. For example, see isTransactionAllowed()

As an example, a school might have a policy that if a student has bounced more than two checks in the last year, they may no longer pass a check to the institution. In this case, a flag would be set each time a bad check were passed to the institution. When getAccountBlockStatus() is called on this account, the rule would look for two or more incidences of the bad check flag on the account, and if present, return a no checks block. A school may reduce this limit to one bad check for accounts that have been overdue, and this logic would also be part of these rules. The system might also turn off certain features during this check, for example, if an account is overdue, the system may turn on certain user-preference privileges.

getAch (accountId)

Get ACH looks into the AccountProtectedInformation class (which triggers a system event) to look for the ACH information for the user. By default, this is stored as a simple string, starting with C for checking or S for savings, followed by a colon, followed by the nine-digit routing number, followed by a colon, followed by the 4-20 digit account number. If this string is available and valid, the service will return an Ach object containing these details.


Get refund.ach.type
Verify that AccountProtectedInfo.bankType matchest refund.ach.type
Throw “NoAchInformation” exception.
get bankInformation
Is first letter a C or an S?
Throw AccountTypeDoesNotExist exception.
Set accountType to this value.
Is this folowed by :#########: (colon, 9 numbers, colon)
Ach
Throw RoutingNumberMalformed exception.
set abaRoutingNumber to the nine digits.
Is the remainder of the bankDetail numerical and between 4-20 digits long?
set accountNumber to the remaining digits.
Throw AccountNumberMalformed exception.
Return the Ach object.

Class Helpers for Account.

Note that all three account helper types (Name, ElectronicContact and PostalAddress) have a setDefault() method, which will look at all the helpers of the same type associated with an account, and ensure that they are false, before setting the specific helper to the default.

addName (accountIdentifier, Name name)

Check doesKsaAccountExist(accountIdentifier)

Add the name to the account. If the Name.kimNameType already exists, overwrite that name, otherwise add it as a new name. If there is no default name, set the default flag on this name. If the isDefault flag is set, then make this the default name

Note that if there is a name stored, then one of them has to be the default name, so even if a name is passed with isDefault=false, if there is no other default name, this entry must become the default name.

addPostalAddress (accountIdentifier, PostalAddress address)

Check doesKsaAccountExist(accountIdentifier)

Add the address to the account. If the PostalAddress.kimAddressType already exists, overwrite that name, otherwise add it as a new address. If there is no default address, set the default flag on this address. If the isDefault flag is set, then make this the default address.

Note that if there is an address stored, then one of them has to be the default address, so even if an address is passed with isDefault=false, if there is no other default address, this entry must become the default address.

addElectronicContact (accountIdentifier, ElectronicContact contact)

Check doesKsaAccountExist(accountIdentifier)

Add the contact to the account. If the ElectronicContact.kimEmailAddressType/ kimPhoneNumberType already exists, overwrite that field, otherwise add it as a new ElectronicContact. If there is no default contact, set the default flag on this one. If the isDefault flag is set in the passed contact, then make this the default one.

Guest Account Services (Special Cases)

permitGuestAccess (accountToAccess, accountToGrant)

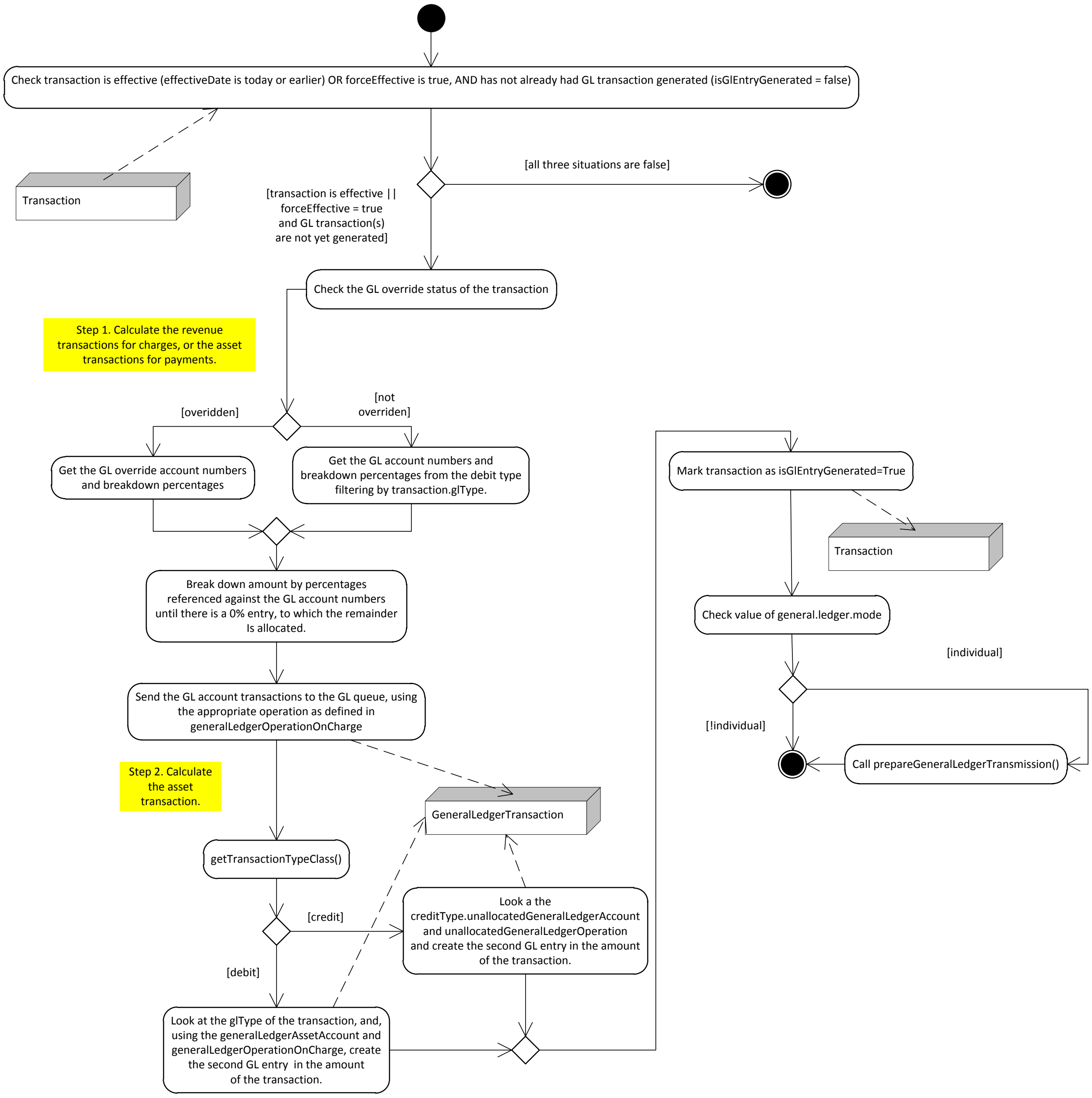
Check that the logged in account is the accountToAccess, and that the accountToGrant exists. Add the accountToAccess, the entity ID of the user and the current date/time to the accountPermission array of the account referenced in accountToGrant.

Transaction Service

makeEffective(transactionId)

makeEffective (transactionId, forceEffective)

Moving a transaction from a pre-effective state to an effective state. Once a transaction is effective, its general ledger entries are created. In certain cases, a transaction might be moved to an effective state before its effective date, in which case, forceEffective is passed as true.


Check transaction is effective (effectiveDate is today or earlier) OR forceEffective is true, AND has not already had GL transaction generated (isGlEntryGenerated = false)
Transaction
[all three situations are false]
[transaction is effective | | forceEffective = true and GL transaction(s) are not yet generated]
Check the GL override status of the transaction
Step 1. Calculate the revenue transactions for charges, or the asset transactions for payments.
[overidden]
[not overriden]
Get the GL override account numbers and breakdown percentages
Get the GL account numbers and breakdown percentages from the debit type filtering by transaction.glType.
Break down amount by percentages referenced against the GL account numbers until there is a 0% entry, to which the remainder Is allocated.
Send the GL account transactions to the GL queue, using the appropriate operation as defined in generalLedgerOperationOnCharge
Step 2. Calculate the asset transaction.
GeneralLedgerTransaction
getTransactionTypeClass()
[credit]
[debit]
Look a the creditType.unallocatedGeneralLedgerAccount and unallocatedGeneralLedgerOperation and create the second GL entry in the amount of the transaction.
Look at the glType of the transaction, and, using the generalLedgerAssetAccount and generalLedgerOperationOnCharge, create the second GL entry in the amount of the transaction.
Mark transaction as isGlEntryGenerated=True
Transaction
Check value of general.ledger.mode
[individual]
[!individual]
Call prepareGeneralLedgerTransmission()

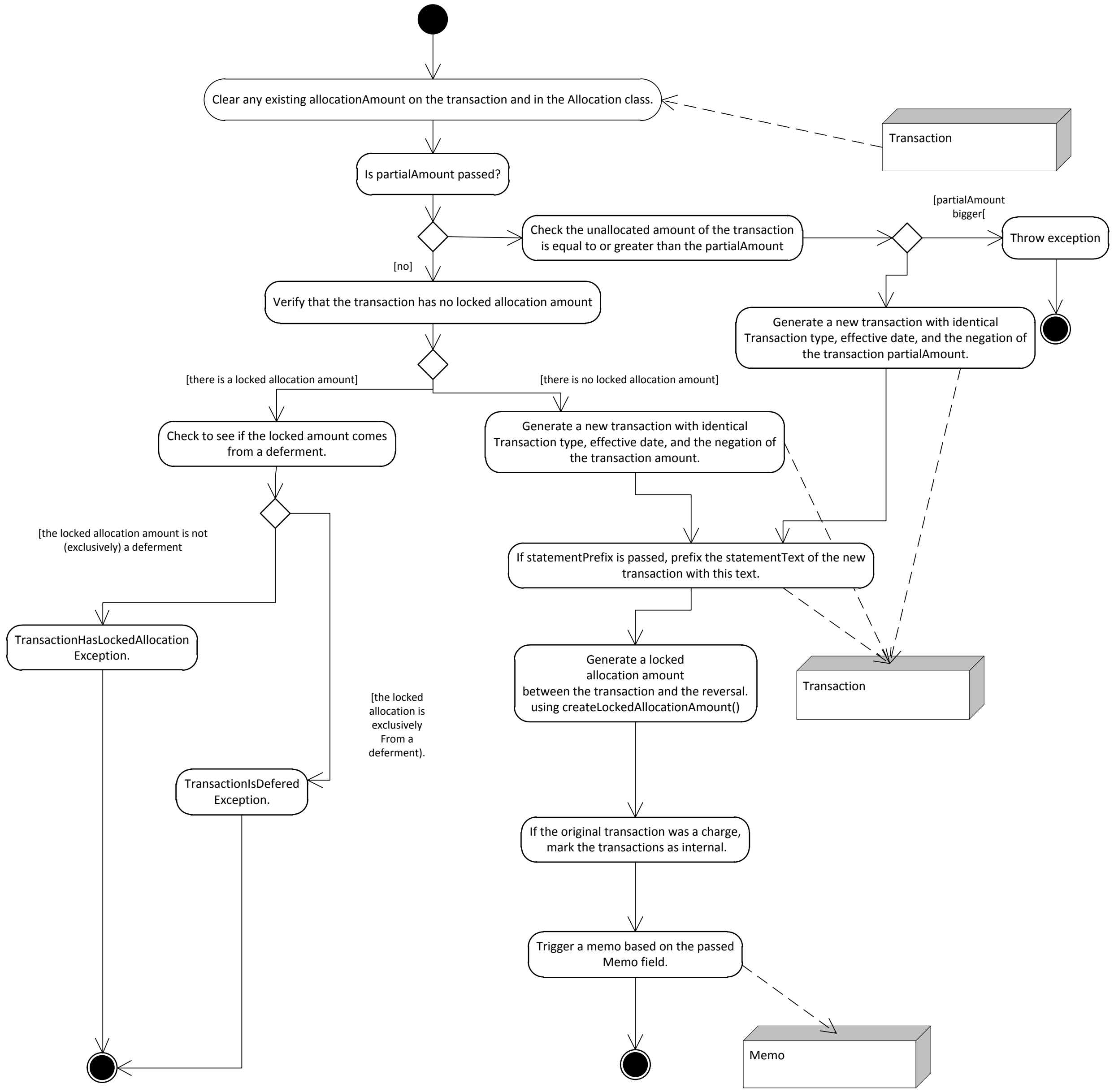
reverseTransaction(transactionId, memo)

reverseTransaction(transactionId, memo, partialAmount)

reverseTransaction(transactionId, memo, statementPrefix)

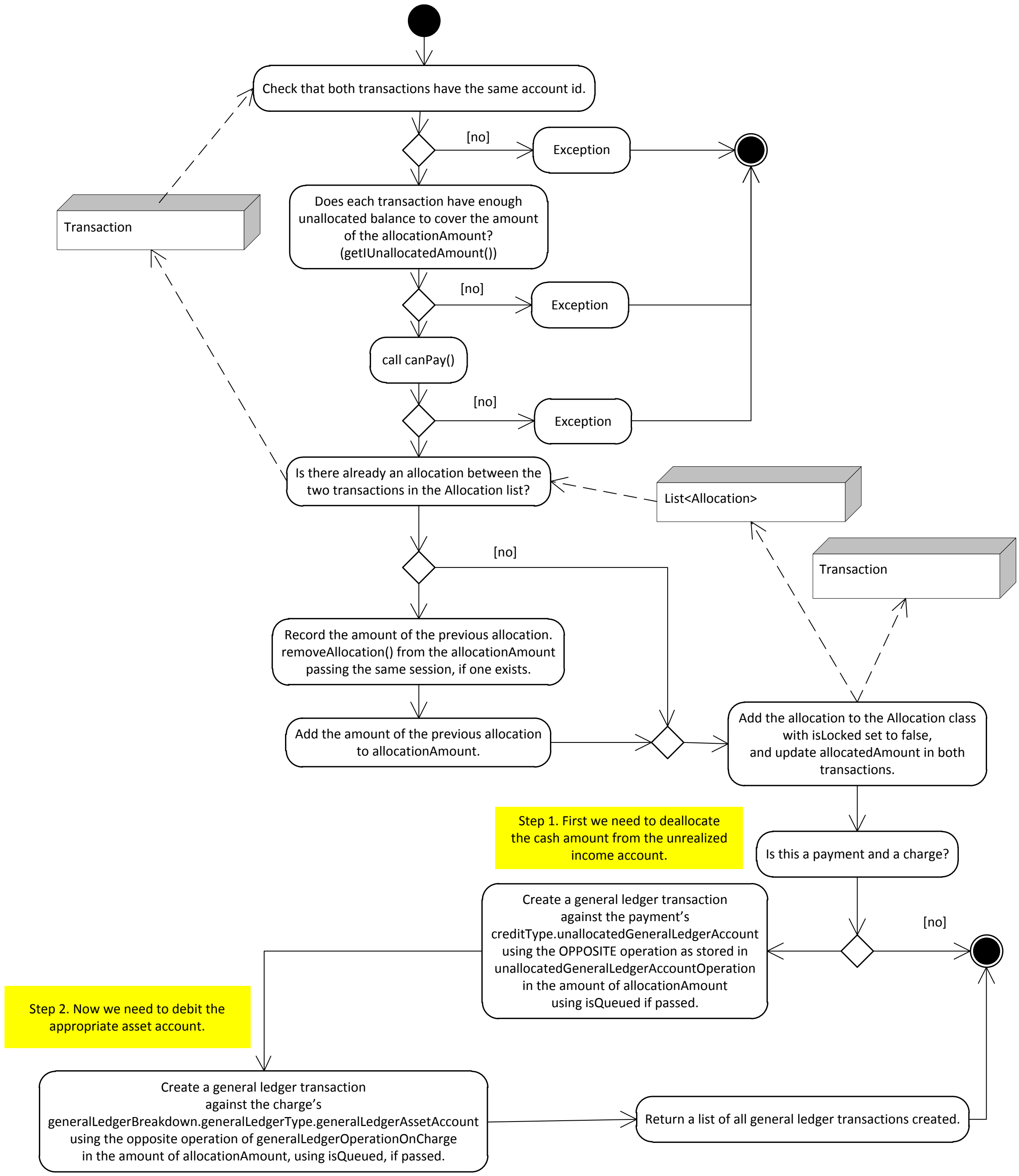
reverseTransaction(transactionId, memo, partialAmount, statementPrefix)

This is used to "reverse" a transaction, which it does by issuing an identical transaction with a negated amount. A memo is passed to the reverse operation, which is entered into the memo field.


Clear any existing allocationAmount on the transaction and in the Allocation class.
Transaction
Is partialAmount passed?
[partialAmount bigger[
Check the unallocated amount of the transaction is equal to or greater than the partialAmount
Throw exception
[no]
Verify that the transaction has no locked allocation amount
Generate a new transaction with identical Transaction type, effective date, and the negation of the transaction partialAmount.
[there is a locked allocation amount]
[there is no locked allocation amount]
Check to see if the locked amount comes from a deferment.
Generate a new transaction with identical Transaction type, effective date, and the negation of the transaction amount.
[the locked allocation amount is not (exclusively) a deferment
If statementPrefix is passed, prefix the statementText of the new transaction with this text.
TransactionHasLockedAllocation Exception.
Generate a locked allocation amount between the transaction and the reversal. using createLockedAllocationAmount()
Transaction
[the locked allocation is exclusively From a deferment).
TransactionIsDefered Exception.
If the original transaction was a charge, mark the transactions as internal.
Trigger a memo based on the passed Memo field.
Memo

createAllocation (transaction1, transaction2, allocationAmount)

createAllocation (transaction1, transaction2, allocationAmount, isQueued)

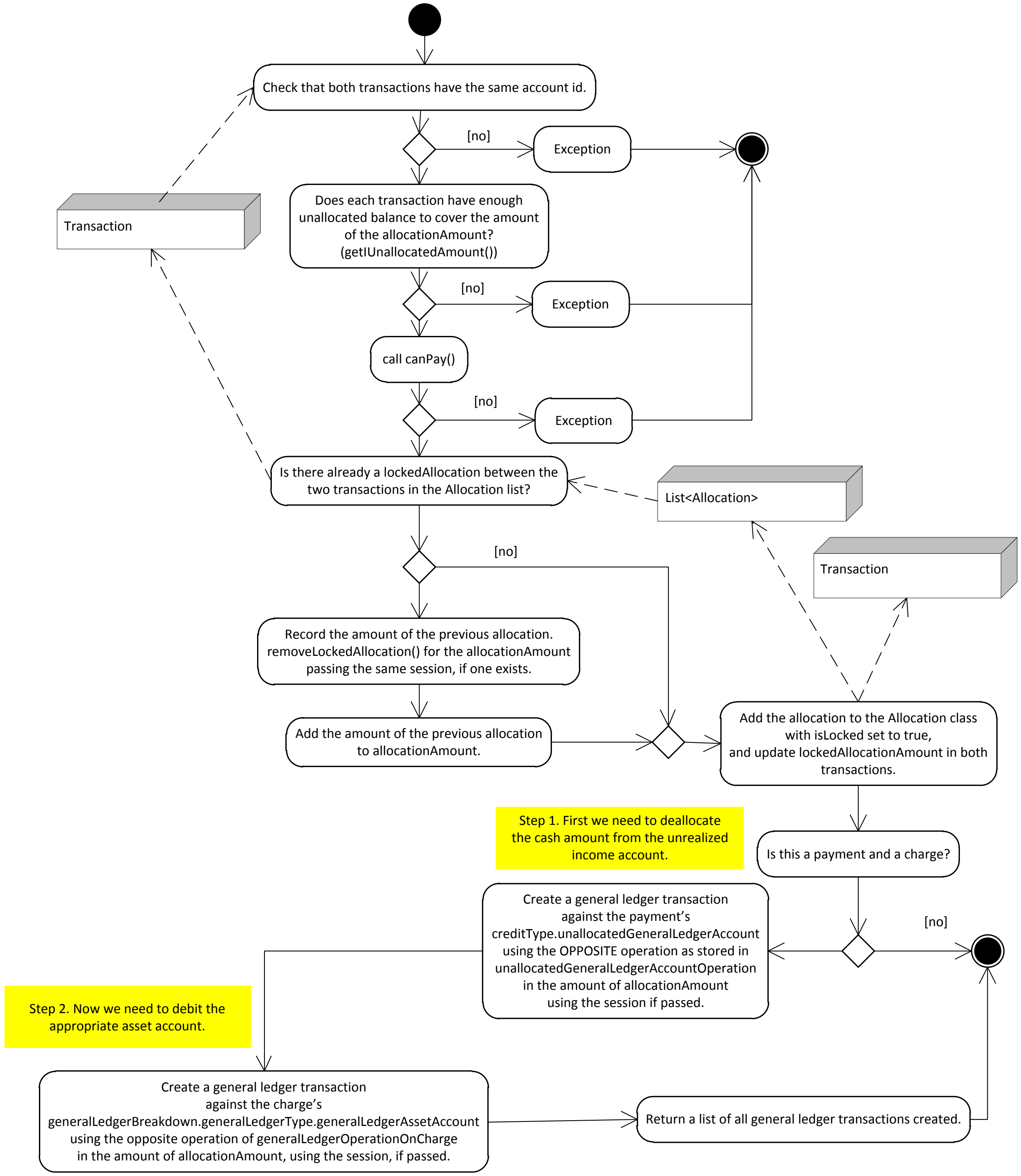

Check that both transactions have the same account id.
[no]
Exception
Transaction
Does each transaction have enough unallocated balance to cover the amount of the allocationAmount? (getIUnallocatedAmount())
[no]
Exception
call canPay()
[no]
Exception
Is there already an allocation between the two transactions in the Allocation list?
List<Allocation>
[no]
Transaction
Record the amount of the previous allocation. removeAllocation() from the allocationAmount passing the same session, if one exists.
Add the amount of the previous allocation to allocationAmount.
Add the allocation to the Allocation class with isLocked set to false, and update allocatedAmount in both transactions.
Is this a payment and a charge?
[no]
Create a general ledger transaction against the payment's creditType.unallocatedGeneralLedgerAccount using the OPPOSITE operation as stored in unallocatedGeneralLedgerAccountOperation in the amount of allocationAmount using isQueued if passed.
Create a general ledger transaction against the charge's generalLedgerBreakdown.generalLedgerType.generalLedgerAssetAccount using the opposite operation of generalLedgerOperationOnCharge in the amount of allocationAmount, using isQueued, if passed.
Return a list of all general ledger transactions created.

Step 1. First we need to deallocate the cash amount from the unrealized income account.

Step 2. Now we need to debit the appropriate asset account.

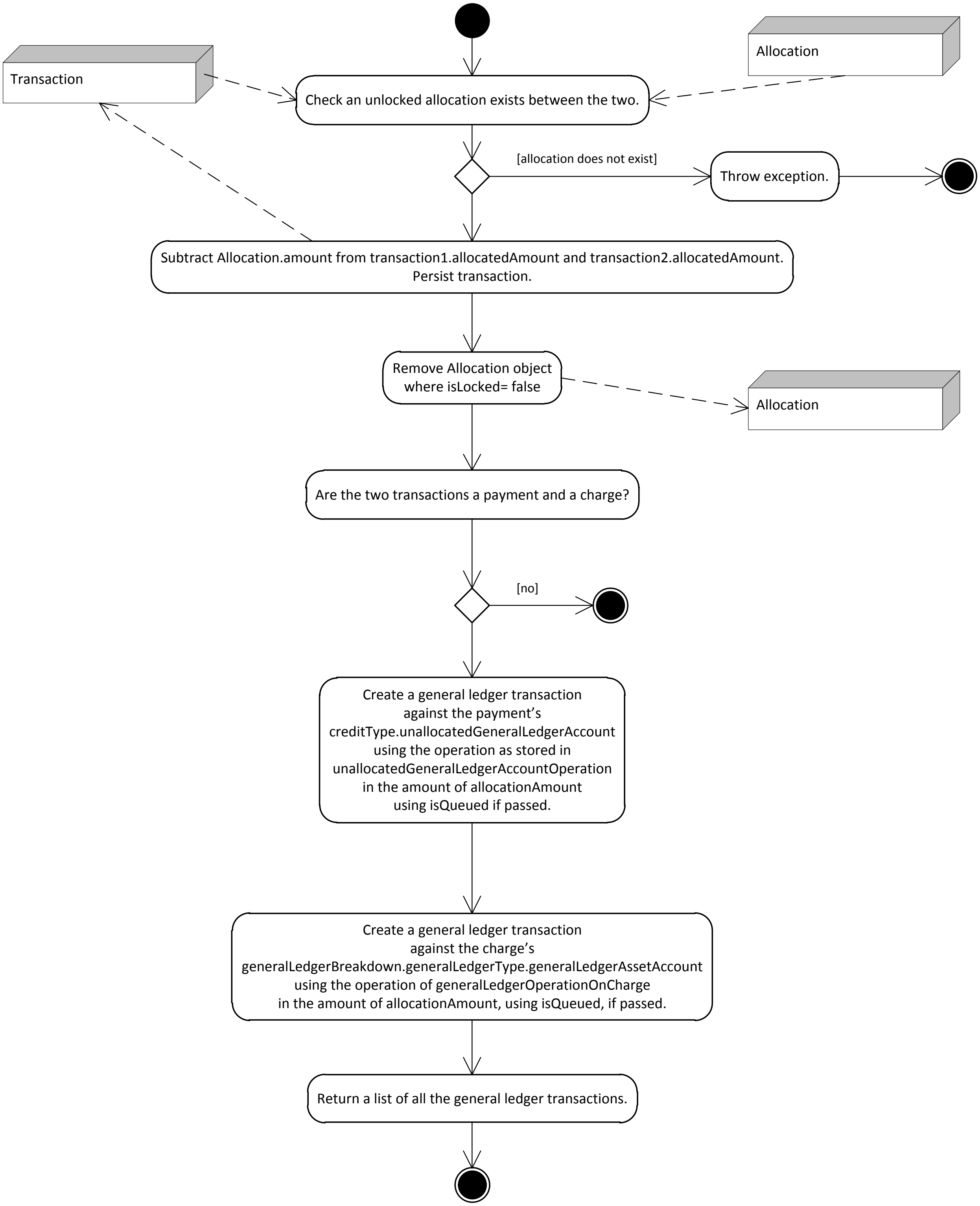
createLockedAllocation (transaction1, transaction2, allocationAmount)

createLockedAllocation (transaction1, transaction2, allocationAmount, isQueued)


Check that both transactions have the same account id.
[no]
Exception
Transaction
Does each transaction have enough unallocated balance to cover the amount of the allocationAmount? (getIUnallocatedAmount())
[no]
Exception
call canPay()
[no]
Exception
Is there already a lockedAllocation between the two transactions in the Allocation list?
List<Allocation>
[no]
Transaction
Record the amount of the previous allocation. removeLockedAllocation() for the allocationAmount passing the same session, if one exists.
Add the amount of the previous allocation to allocationAmount.
Add the allocation to the Allocation class with isLocked set to true, and update lockedAllocationAmount in both transactions.
Step 1. First we need to deallocate the cash amount from the unrealized income account.
Is this a payment and a charge?
Create a general ledger transaction against the payment's creditType.unallocatedGeneralLedgerAccount using the OPPOSITE operation as stored in unallocatedGeneralLedgerAccountOperation in the amount of allocationAmount using the session if passed.
[no]
Step 2. Now we need to debit the appropriate asset account.
Create a general ledger transaction against the charge's generalLedgerBreakdown.generalLedgerType.generalLedgerAssetAccount using the opposite operation of generalLedgerOperationOnCharge in the amount of allocationAmount, using the session, if passed.
Return a list of all general ledger transactions created.

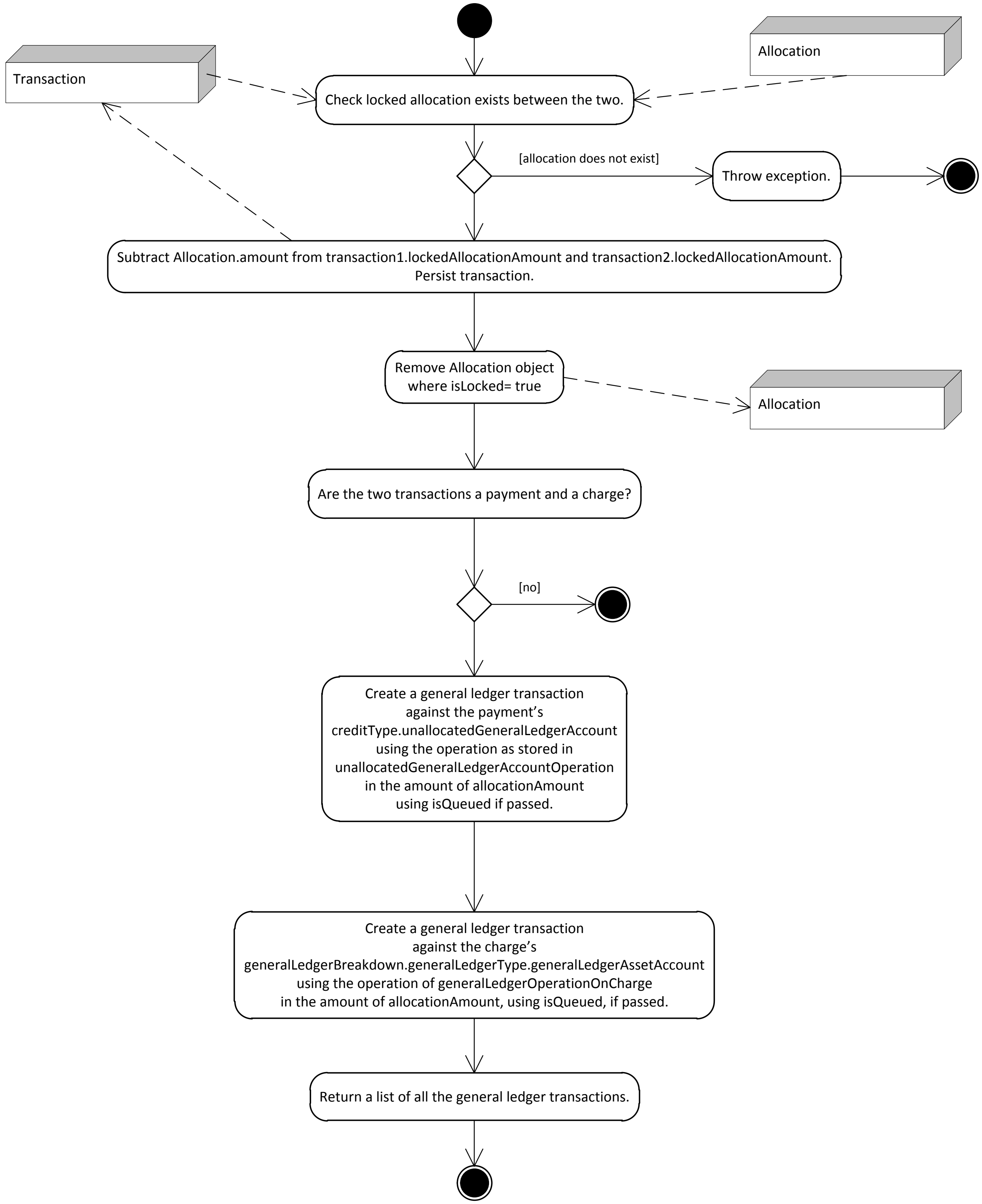
removeAllocation (transactionId1, transactionId2)

removeAllocation (transactionId1, transactionId2, isQueued)


Transaction
Check an unlocked allocation exists between the two.
Allocation
[allocation does not exist]
Throw exception.
Subtract Allocation.amount from transaction1.allocatedAmount and transaction2.allocatedAmount.
Persist transaction.
Remove Allocation object
where isLocked= false
Allocation
Are the two transactions a payment and a charge?
[no]
Create a general ledger transaction
against the payment’s
creditType.unallocatedGeneralLedgerAccount
using the operation as stored in
unallocatedGeneralLedgerAccountOperation
in the amount of allocationAmount
using isQueued if passed.
Create a general ledger transaction
against the charge’s
generalLedgerBreakdown.generalLedgerType.generalLedgerAssetAccount
using the operation of generalLedgerOperationOnCharge
in the amount of allocationAmount, using isQueued, if passed.
Return a list of all the general ledger transactions.

removeLockedAllocation (transactionId1, transactionId2)

removeLockedAllocation (transactionId1, transactionId2, isQueued)


Transaction
Allocation
Check locked allocation exists between the two.
[allocation does not exist]
Throw exception.
Subtract Allocation.amount from transaction1.lockedAllocationAmount and transaction2.lockedAllocationAmount. Persist transaction.
Remove Allocation object where isLocked= true
Allocation
Are the two transactions a payment and a charge?
[no]
Create a general ledger transaction against the payment's creditType.unallocatedGeneralLedgerAccount using the operation as stored in unallocatedGeneralLedgerAccountOperation in the amount of allocationAmount using isQueued if passed.
Create a general ledger transaction against the charge's generalLedgerBreakdown.generalLedgerType.generalLedgerAssetAccount using the operation of generalLedgerOperationOnCharge in the amount of allocationAmount, using isQueued, if passed.
Return a list of all the general ledger transactions.

removeAllAllocations (transactionId)

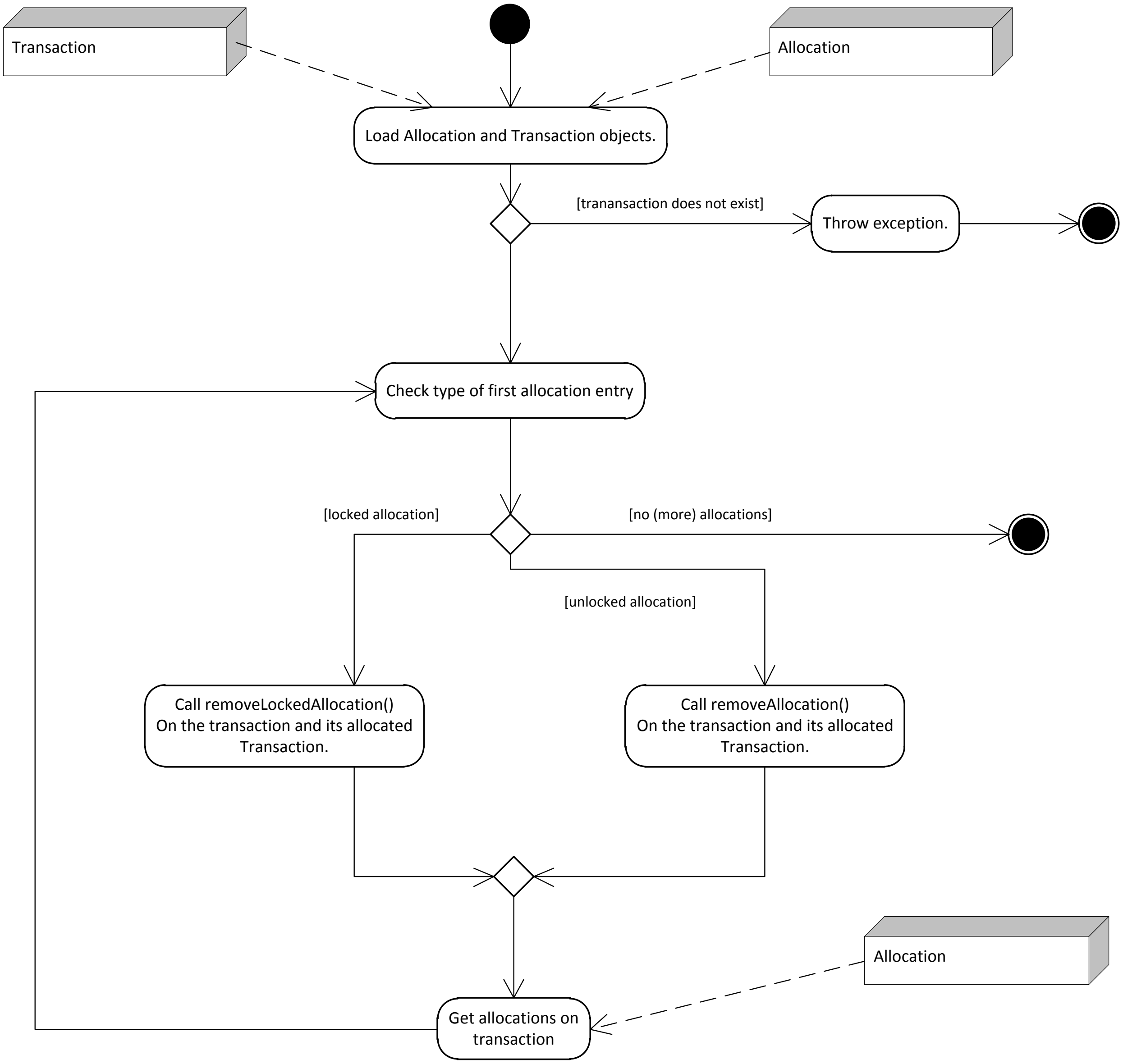
removeAllAllocations (transactionId, isQueued)

This method would most often be used when a payment has “bounced” and needs to be reversed off the account. A search is made of all allocations made against the transaction, and then those allocations are reversed. It is expected that the transaction referenced would then be reversed before a new payment application is applied to the account which would reallocate it. Pass isQueued to each change in allocation, if passed.

Return a list of all general ledger transactions created.

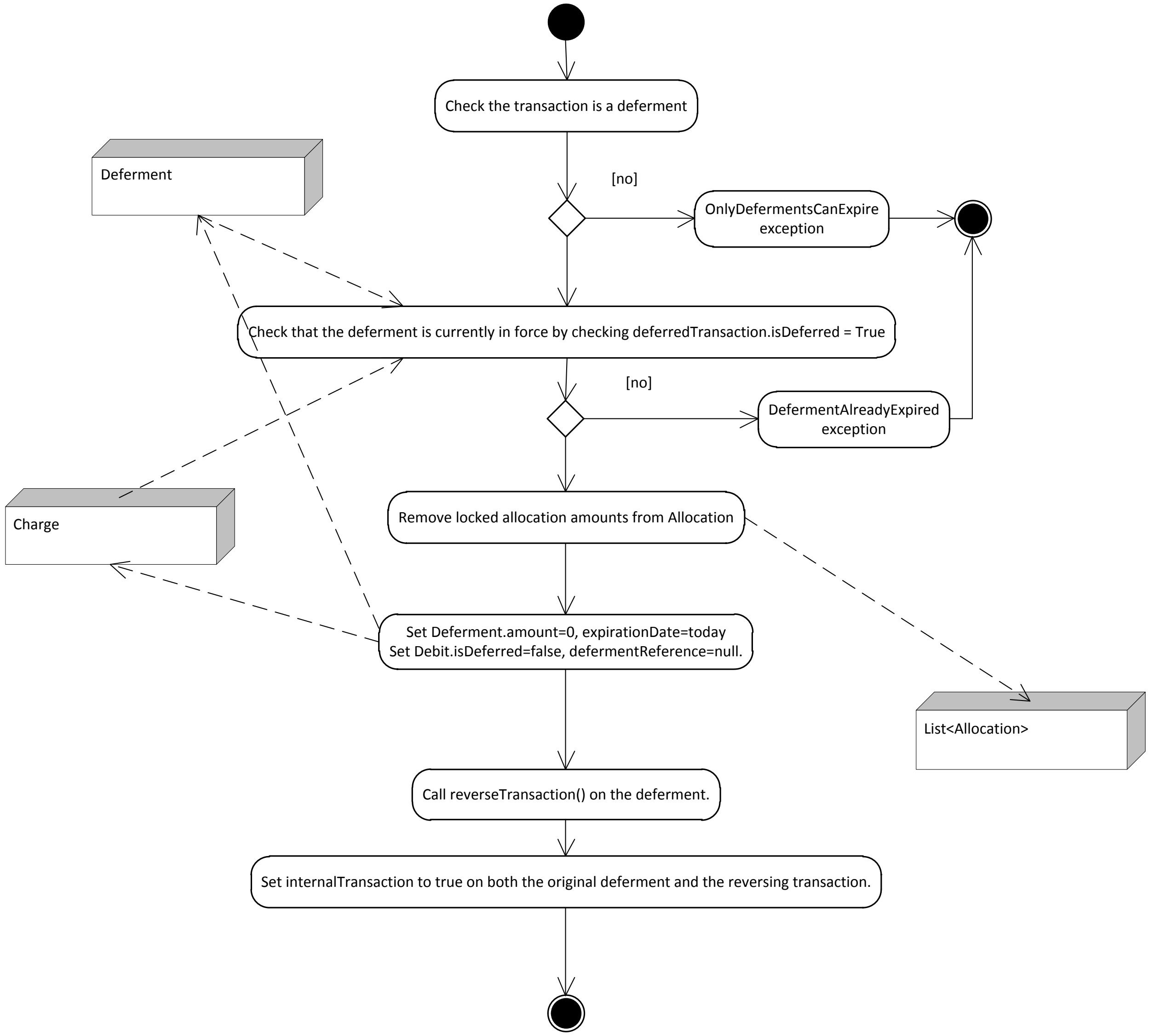
getUnallocatedAmount (transactionId)

Get the amount of the transaction, and subtract allocatedAmount and lockedAllocationAmount. Return this value.


Transaction
Allocation
Load Allocation and Transaction objects.
[tranansaction does not exist]
Throw exception.
Check type of first allocation entry
[locked allocation]
[no (more) allocations]
[unlocked allocation]
Call removeLockedAllocation()
On the transaction and its allocated
Transaction.
Call removeAllocation()
On the transaction and its allocated
Transaction.
Allocation
Get allocations on
transaction

expireDeferment(transactionId)

expireDeferment() is a method of Deferment.


Check the transaction is a deferment
[no]
OnlyDefermentsCanExpire exception
Deferment
Check that the deferment is currently in force by checking deferredTransaction.isDeferred = True
[no]
DefermentAlreadyExpired exception
Charge
Remove locked allocation amounts from Allocation
Set Deferment.amount=0, expirationDate=today
Set Debit.isDeferred=false, defermentReference=null.
List<Allocation>
Call reverseTransaction() on the deferment.
Set internalTransaction to true on both the original deferment and the reversing transaction.

canPay (transaction1, transaction2)

canPay (transaction1, transaction2, priority)

canPay(transaction1, transaction2, priorityFrom, priorityTo)

Check both transactions are on the same account. If not, return false.

Check the transactions are compatible:

Transaction 1	Transaction 2

Payment with positive amount	Charge with positive amount
Payment with positive amount	Payment with negative amount (payment reversal)
Charge with positive amount	Charge with negative amount (charge reversal)

If not, return false.

If the two transactions are charges, then return true.

If one transaction is a payment, look into its creditType.permissableDebitArray[]

Use standard wildcard matching between the members of the permissableDebitArray[] to see if any of them match against the debitType of the Charge. If a priority is passed, only those types that match the priority or range or priorities will be checked.

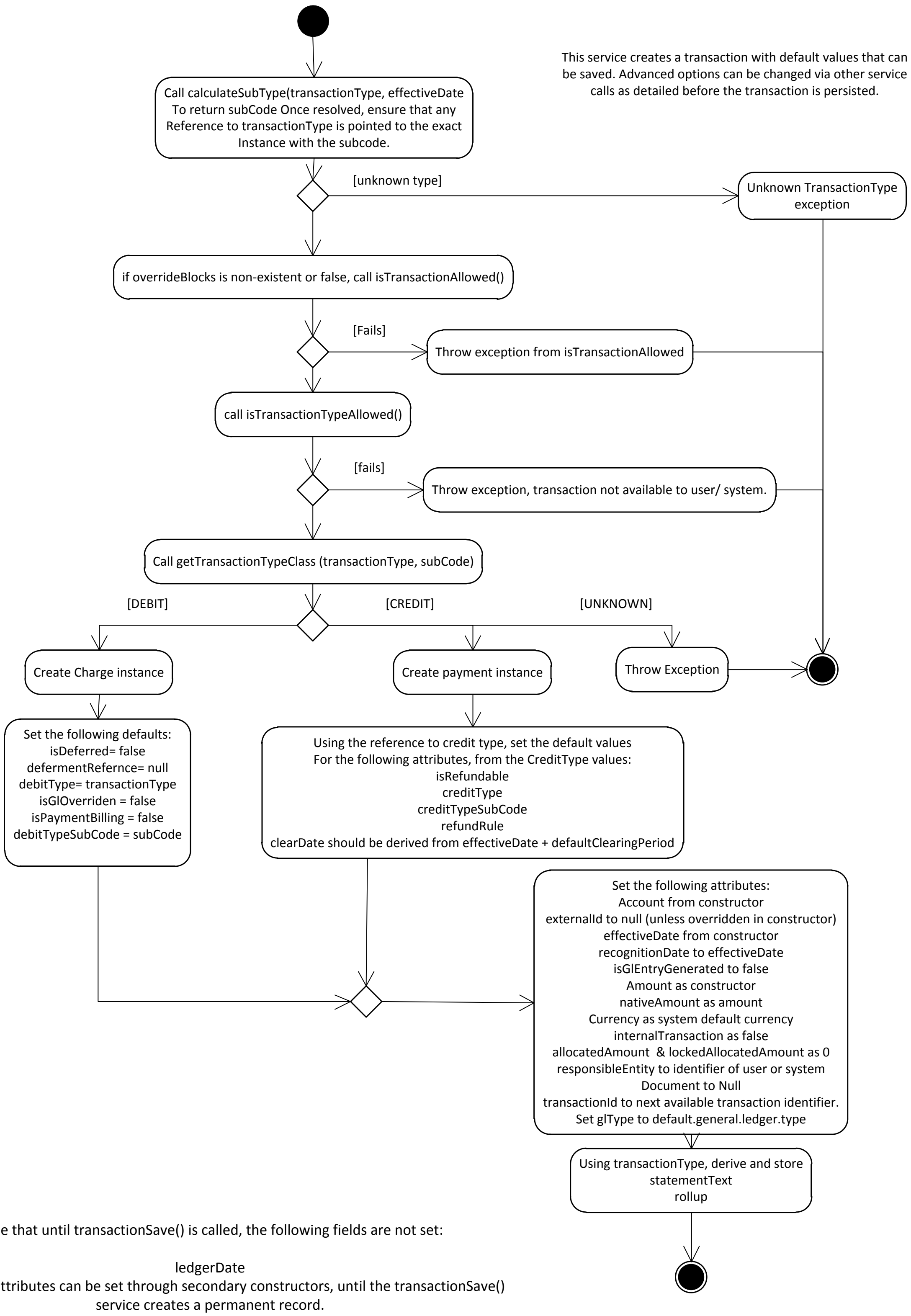
If there is a match, return true, otherwise return false.

createTransaction (transactionType, account, effectiveDate, amount)

createTransaction (transactionType, account, effectiveDate, amount externalId)

createTransaction (transactionType, account, effectiveDate, amount, overrideBlocks)

createTransaction (transactionType, account, effectiveDate, amount, externalId, overrideBlocks)


Call calculateSubType(transactionType, effectiveDate
To return subCode Once resolved, ensure that any
Reference to transactionType is pointed to the exact
Instance with the subcode.
[unknown type]
Unknown TransactionType
exception
if overrideBlocks is non-existent or false, call isTransactionAllowed()
[Fails]
Throw exception from isTransactionAllowed
call isTransactionTypeAllowed()
[fails]
Throw exception, transaction not available to user/ system.
Call getTransactionTypeClass (transactionType, subCode)
[DEBIT]
[CREDIT]
[UNKNOWN]
Create Charge instance
Create payment instance
Throw Exception
Set the following defaults:
isDeferred= false
defermentRefernce= null
debitType= transactionType
isGlOverriden = false
isPaymentBilling = false
debitTypeSubCode = subCode
Using the reference to credit type, set the default values
For the following attributes, from the CreditType values:
isRefundable
creditType
creditTypeSubCode
refundRule
clearDate should be derived from effectiveDate + defaultClearingPeriod
Set the following attributes:
Account from constructor
externalId to null (unless overridden in constructor)
effectiveDate from constructor
recognitionDate to effectiveDate
isGlEntryGenerated to false
Amount as constructor
nativeAmount as amount
Currency as system default currency
internalTransaction as false
allocatedAmount & lockedAllocatedAmount as 0
responsibleEntity to identifier of user or system
Document to Null
transactionId to next available transaction identifier.
Set glType to default.general.ledger.type
Using transactionType, derive and store
statementText
rollup

This service creates a transaction with default values that can be saved. Advanced options can be changed via other service calls as detailed before the transaction is persisted.

Note that until transactionSave() is called, the following fields are not set:

ledgerDate
Many other attributes can be set through secondary constructors, until the transactionSave() service creates a permanent record.

getTransactionTypeClass (transactionTypeId)

For a given transactionType, return CreditType.class, DebitType.class or null (no match found).

calculateSubType (transactionType, effectiveDate)

Look through the TransactionType objects to find the exact subcode that refers to the date and transactionType in question. Return the subcode, so that this can be used as the second part of the key when referencing the specific transaction type.

saveTransaction (transaction)

Set ledgerDate to current system date. Persist the transaction. It should be noted that a transaction with a ledgerDate set to a non-null value has been saved, and therefore many of its attributes are immutable. If the transaction is a payment, with a tag of cash.tracking.tag, call the checkCashLimit() method.

setExternalId (transactionId, externalId)

Check the transactionId exists, otherwise throw exception.

Check the transaction has not yet been saved, otherwise throw exception.

Set the externalId.

setOriginationDate (transactionId, originationDate)

Check the transactionId exists, otherwise throw exception.

Check the transaction has not yet been saved, otherwise throw exception.

Set the originationDate.

setForeignTransaction (transactionId, currency, nativeAmount)

Check the transactionId exists, otherwise throw exception.

Check the transaction has not yet been saved, otherwise throw exception.

Check the currency exists, otherwise throw an exception.

Set the currency and nativeAmount.

setStatementText (transactionId, statementText)

Check the transactionId exists, otherwise throw exception.

Set the statementText.

setRollup (transactionId, rollup)

Check the transactionId exists, otherwise throw exception.

Check the rollup exists, otherwise thrown an exception.

Set the rollup.

setDocument (transactionId, document)

//STILL ON THE TODO LIST.

Check the transactionId exists, otherwise throw exception.

Check the transaction has not yet been saved, otherwise throw exception.

Set the document.

setGlOverride (transactionId, glOverride)

Note that the GL can be overridden in one of two ways. This method permits the passing of a GlOverride object, which is a "one off" override. The second method passes a breakdownType, which requires the transaction type to have more than one GeneralLedgerBreakdown objects. All default GL breakdowns have a breakdownType of 0. Other ones may be added, to permit a more flexible typing of GL Breakdowns. For example, transferrin to third-party accounts may require a completely different GL breakdown.

Check the transactionId exists, otherwise throw exception.

Check the transactionId is referencing a Debit or child of Debit.

Check the transaction has not yet been saved, otherwise throw exception.

Set isGlOverriden to true, and set glOverrideArray to glOverride

setGlOverride (transactionId, breakdownType)

Check the transactionId exists, otherwise throw exception.

Check the transactionId is referencing a Debit or child of Debit.

Check the transaction has not yet been saved, otherwise throw exception.

Check the transaction type has a GeneralLedgerBreakdown of the type passed in the parameters. Otherwise thrown exception

Set isGlOverriden to true.

Copy the contents of the GeneralLedgerBreakdown there breakdownType is equal to the breakdownType referenced in the parameter to the transaction->glOverrideArray.

setGlType (transactionId, newGlType)

By default, the generalLedgerType of a transaction is set to default.general.ledger.type (passed via the code attribute). The generalLedgerType of a transaction can be altered before the transaction is saved to the KSA system. This type effects the way in which general ledger transactions are calculated.

setIsRefundable (transactionId, isRefundable)

Check the transactionId exists, otherwise throw exception.

Check the transactionId is referencing a Credit or child of Credit.

Check the transaction has not yet been saved, otherwise throw exception.

Set isRefundable.

setRefundRule (transactionId, refundRule)

//STILL ON TODO LIST.

Check the transactionId exists, otherwise throw exception.

Check the transactionId is referencing a Credit or child of Credit.

Check the transaction has not yet been saved, otherwise throw exception.

Set refundRule.

setOriginalAmount (transactionId, originalAmount)

Check the transactionId exists, otherwise throw exception.

Check the transactionId is referencing a Deferment.

Check the transaction has not yet been saved, otherwise throw exception.

Set originalAmount.

setExpirationDate (transactionId, originalAmount)

Check the transactionId exists, otherwise throw exception.

Check the transactionId is referencing a Deferment.

Set expirationDate.

setTransactionToBeDeferred (transactionId, Charge chargeToDefer)

//TODO

setClearDate (transactionId, clearDate)

Check the transactionId exists, otherwise throw exception.

Check the transactionId is referencing a Payment.

Check the transaction has not yet been saved, otherwise throw exception.

Set clearDate.

setIsPaymentBilling (transactionId, isTrue)

If the transaction is a charge, then set the isPaymentBilling to isTrue.

setDerivativeTransactionReference (transactionId, pointerTransactionId)

Check transactionId points to a charge, and that pointerTransactionId points to a valid charge. Set transactionId->derivativeTransactionReference to pointerTransactionId.

clearDerivativeTransactionReference (transactionId)

Set transactionId->derivativeTransactionReference to null.

saveToKsaLedger (transactionId)

This actually saves the transaction out to the database, and locks many of the attributes of the transaction. Set ledgerDate to currentDate and save to database.

getDaysBeforeDueDate()

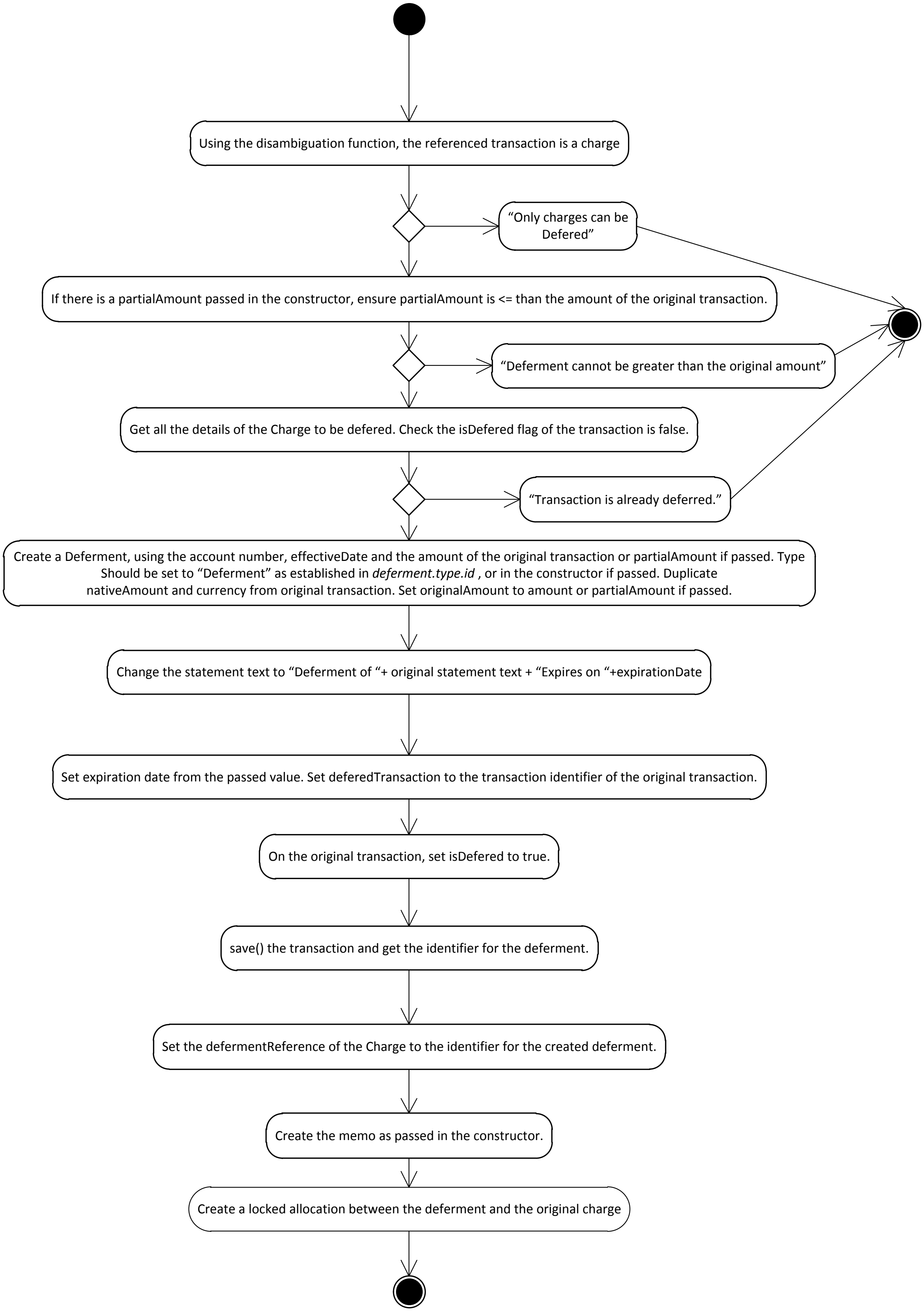
Returns the number of days between now and the due date (effectiveDate) of the transaction.

defer(transactionIdtoDefer, expirationDateOfDeferment, memo)

defer(transactionIdtoDefer, expirationDateOfDeferment, memo, partialAmount)

defer(transactionIdtoDefer, expirationDateOfDeferment, memo, transactionTypeOverride)

defer(transactionIdtoDefer, expirationDateOfDeferment, memo, partialAmount, transactionTypeIdentifierOverride)

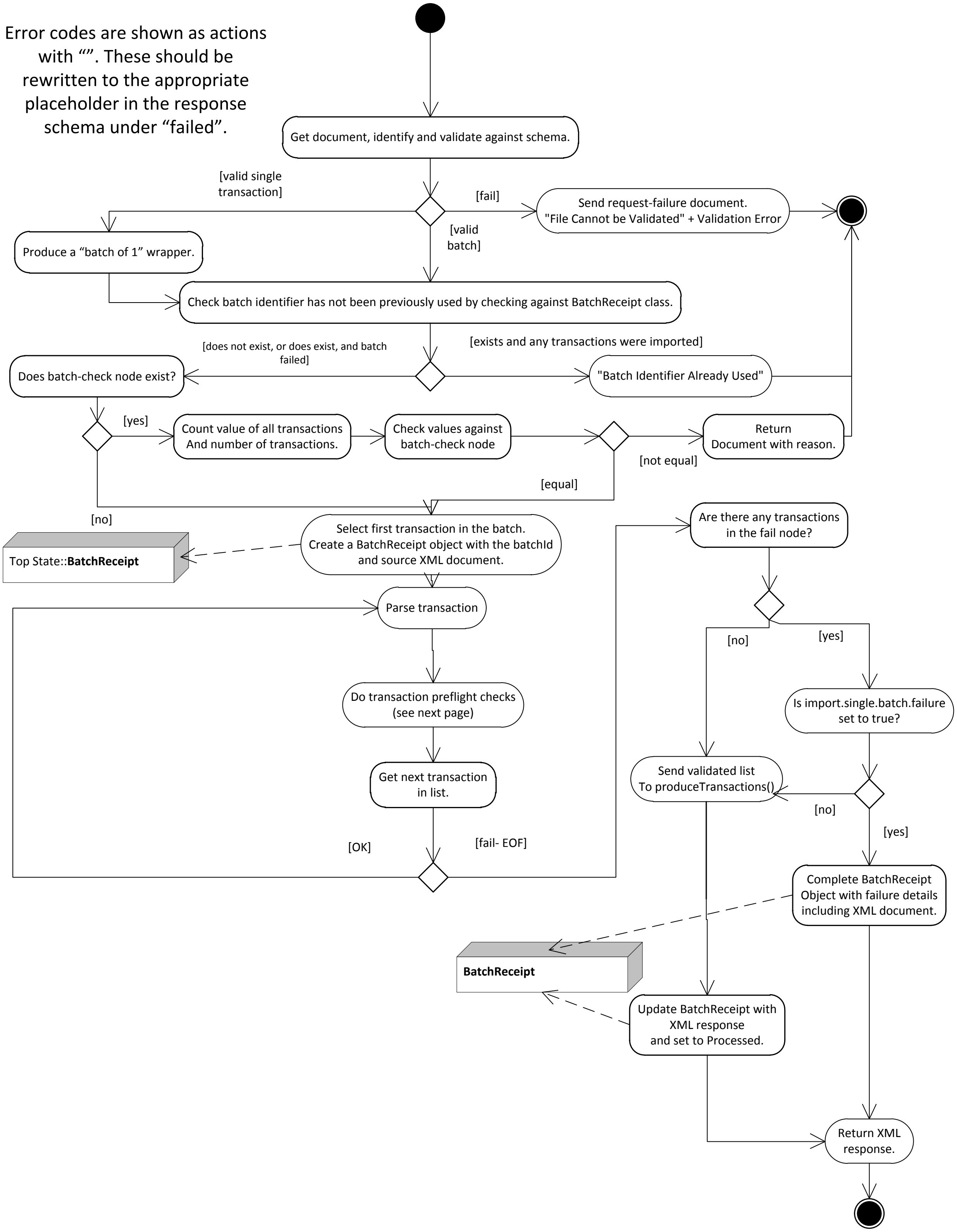

Using the disambiguation function, the referenced transaction is a charge
“Only charges can be Defered”
If there is a partialAmount passed in the constructor, ensure partialAmount is <= than the amount of the original transaction.
“Deferment cannot be greater than the original amount”
Get all the details of the Charge to be defered. Check the isDefered flag of the transaction is false.
“Transaction is already deferred.”
Create a Deferment, using the account number, effectiveDate and the amount of the original transaction or partialAmount if passed. Type Should be set to “Deferment” as established in *deferment.type.id* , or in the constructor if passed. Duplicate nativeAmount and currency from original transaction. Set originalAmount to amount or partialAmount if passed.
Change the statement text to “Deferment of “+ original statement text + “Expires on “+expirationDate
Set expiration date from the passed value. Set deferedTransaction to the transaction identifier of the original transaction.
On the original transaction, set isDefered to true.
save() the transaction and get the identifier for the deferment.
Set the defermentReference of the Charge to the identifier for the created deferment.
Create the memo as passed in the constructor.
Create a locked allocation between the deferment and the original charge

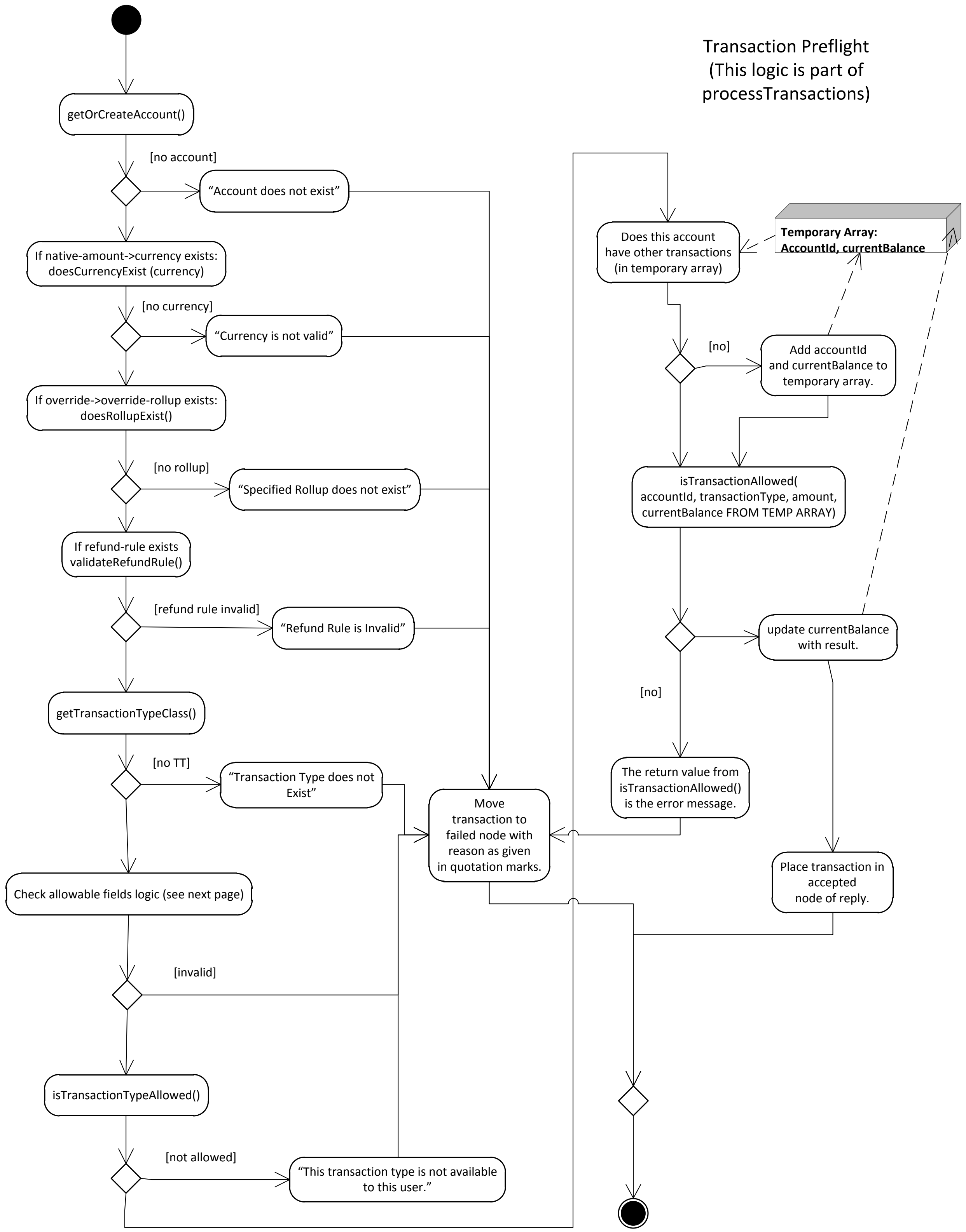
processTransactions (inputFile)

This process is designed to process an XML file of transactions, per the standard schema. It should be recognized that this will be a standard process for many forms of transaction import from external systems. This process system prepares the file to be run through the produceTransactions() service, which will actually create the transactions. This double-stepped process is required to permit schools to have a policy of fail-one-fail-all, or fail individual transactions. It will also be possible in future iterations, for the transmitting system to decide on the failure tolerance of the transactions, subject to system-wide configuration parameters.

If the sending system includes the batch-check node, the system will calculate the total number of transactions and value of all transactions (taken as a literal. Negative transactions will negate, positive transactions will accrue, regardless of charge or payment status). If the batch totals do not match, the system will reject the batch *en masse*.

Error codes are shown as actions with “”. These should be rewritten to the appropriate placeholder in the response schema under “failed”.


Get document, identify and validate against schema.
[valid single transaction]
[fail]
Send request-failure document. "File Cannot be Validated" + Validation Error
Produce a “batch of 1” wrapper.
[valid batch]
Check batch identifier has not been previously used by checking against BatchReceipt class.
[does not exist, or does exist, and batch failed]
[exists and any transactions were imported]
Does batch-check node exist?
"Batch Identifier Already Used"
[yes]
Count value of all transactions And number of transactions.
Check values against batch-check node
Return Document with reason.
[not equal]
[equal]
[no]
Top State::BatchReceipt
Select first transaction in the batch. Create a BatchReceipt object with the batchId and source XML document.
Are there any transactions in the fail node?
Parse transaction
[no]
[yes]
Do transaction preflight checks (see next page)
Is import.single.batch.failure set to true?
Get next transaction in list.
Send validated list To produceTransactions()
[no]
[yes]
[OK]
[fail- EOF]
Complete BatchReceipt Object with failure details including XML document.
BatchReceipt
Update BatchReceipt with XML response and set to Processed.
Return XML response.


Transaction Preflight
(This logic is part of
processTransactions)
getOrCreateAccount()
[no account]
“Account does not exist”
If native-amount->currency exists:
doesCurrencyExist (currency)
[no currency]
“Currency is not valid”
If override->override-rollup exists:
doesRollupExist()
[no rollup]
“Specified Rollup does not exist”
If refund-rule exists
validateRefundRule()
[refund rule invalid]
“Refund Rule is Invalid”
getTransactionTypeClass()
[no TT]
“Transaction Type does not Exist”
Check allowable fields logic (see next page)
[invalid]
isTransactionTypeAllowed()
[not allowed]
“This transaction type is not available to this user.”
Move transaction to failed node with reason as given in quotation marks.
Does this account have other transactions (in temporary array)
Temporary Array:
AccountId, currentBalance
[no]
Add accountId and currentBalance to temporary array.
isTransactionAllowed(
accountId, transactionType, amount,
currentBalance FROM TEMP ARRAY)
update currentBalance with result.
[no]
The return value from isTransactionAllowed() is the error message.
Place transaction in accepted node of reply.

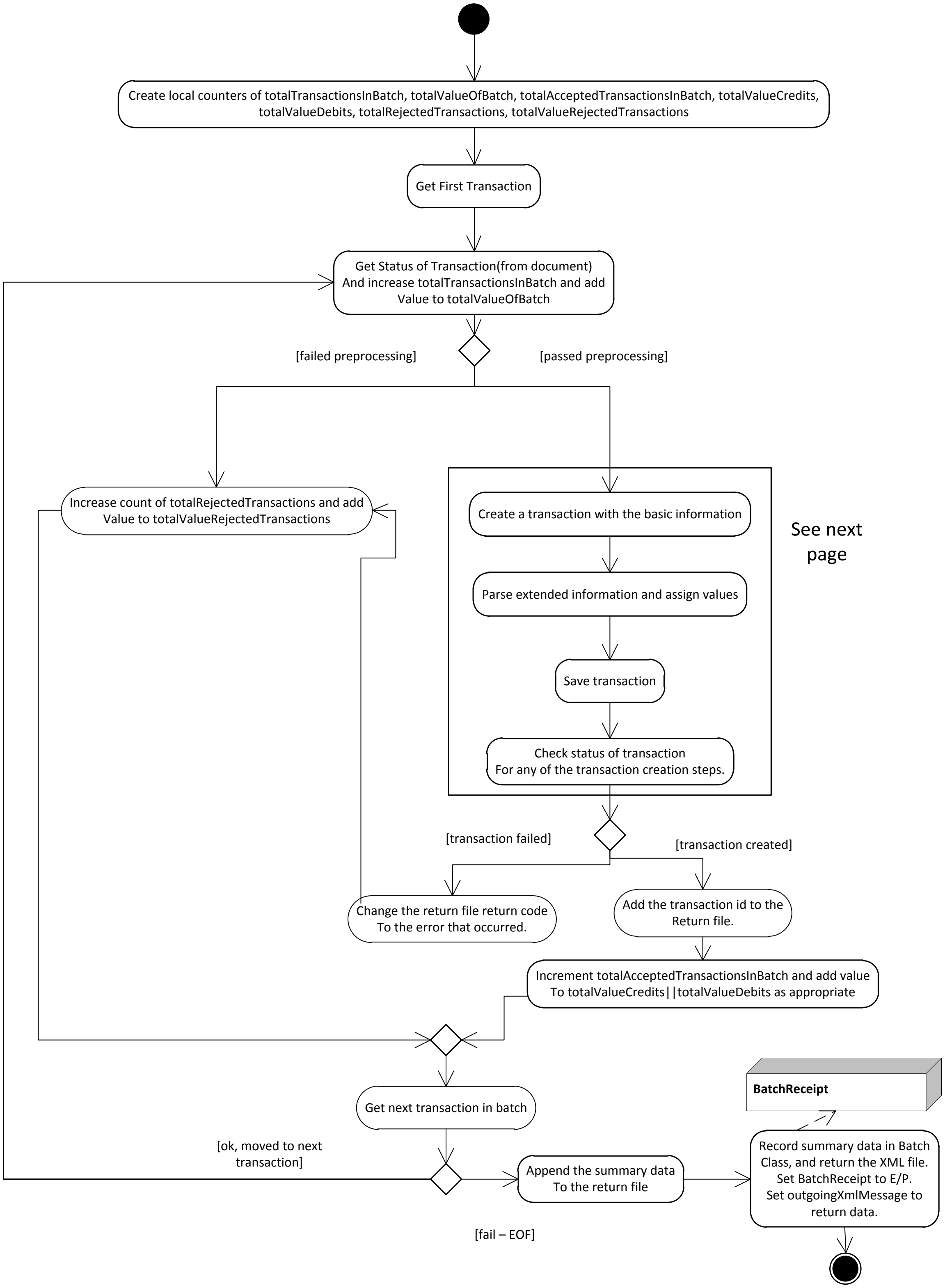
Allowable Fields Logic

For each of the types, the fields below are allowed. If they exist for another type, then the transaction is invalid. For example, if a charge carries a refund rule, the transaction is invalid.

Credit Types (payments)	Debit Types (charges)	Deferments only (not yet supported in schema)
<override-refund-rule>	<general-leger-override>	<expiration-date>
<override-clear-date>		<deferred-transaction>
<override-clear-period>		
<is-refundable>		

produceTransactions(validatedInputFile)

This method is not available as a service, it must be called via the processTransactions() method to ensure that the transactions have all been checked before starting to add them to the KSA ledger.


Create local counters of totalTransactionsInBatch, totalValueOfBatch, totalAcceptedTransactionsInBatch, totalValueCredits, totalValueDebits, totalRejectedTransactions, totalValueRejectedTransactions
Get First Transaction
Get Status of Transaction(from document) And increase totalTransactionsInBatch and add Value to totalValueOfBatch
[failed preprocessing]
[passed preprocessing]
Increase count of totalRejectedTransactions and add Value to totalValueRejectedTransactions
Create a transaction with the basic information
Parse extended information and assign values
Save transaction
Check status of transaction For any of the transaction creation steps.
See next page
[transaction failed]
[transaction created]
Change the return file return code To the error that occurred.
Add the transaction id to the Return file.
Increment totalAcceptedTransactionsInBatch and add value To totalValueCredits||totalValueDebits as appropriate
Get next transaction in batch
[ok, moved to next transaction]
Append the summary data To the return file
[fail – EOF]
BatchReceipt
Record summary data in Batch Class, and return the XML file. Set BatchReceipt to E/P. Set outgoingXmlMessage to return data.

Transaction Creation Flow (Boxed on previous page)

If <incoming-identifier> is set, call:

createTransaction(<transaction-type>, <incoming-identifier>, <account-idenifier>, <effective-date>, <amount>)

Else

createTransaction(<transaction-type>, <account-idenifier>, <effective-date>, <amount>)

This will return the transaction identifier, which should be stored in the reply XML document for this transaction. You can now also get the ledgerDate to be stored as <accepted-date>

If <origination-date> is set,

setOriginationDate (transactionId, <origination-date>)

If <recognition-date> is set

Set recognitionDate to this value

ELSE

Set recognitionPeriod to effectiveDate

If <native-amount>&&<currency> are set,

setForeignTransaction (transactionId, <currency>, <native-amount>)

Else

setForeignTransaction (transactionId, DEFAULT_SYSTEM_CURRENCY, <amount>)

If <document> exists (this will likely become non-optional)

setDocument (transactionId, <document>)

If <override> <is-refundable> is set to true,

setIsRefundable (transactionId, True)

If <override> <is-refundable> is set to false,

setIsRefundable (transactionId, False)

If <override> <refund-rule> is set,

setRefundRule (transactionId, <refund-rule>)

If <override><override-rollup> is set,

setRollup (transactionId, <rollup>) Not that the XML document field matches on the xmlName attribute.

If <override><override-statement-text> is set,

setStatementText (transactionId, <override><override-statement-text>)

If <override><override-clear-date> is set,

setClearDate (transactionId, <clear-date>)

If <override><override-clear-period> is set,

setClearDate (transactionId, current date+ <clear-period>)

If <override><general-ledger-type> is set,

Set the generalLedgerType of the transaction to <general-ledger-type>

ELSE

Default to general.ledger.type

isTransactionAllowed(accountId, transactionType, amount)

isTransactionAllowed (acountId, transactionType, amount, overrideCurrentBalance)

Returns a list of reasons the transaction would fail, or no list if the transaction would work. This validation is performed in **RULES**.

This performs any number of checks, but is expected to cover blocks and credit limit checks.

Using the rules-based block evaluation, (see getAccountBlockedStatus) evaluate if a transaction is allowed to be posted to an account. An initial check for the existence of the account and the transaction type (if passed) is first performed.

This allows the following types of scenarios. This list serves as examples. Schools may have simpler or more complex blocking rules as defined in their own policies. Blocks are defined as AccountBlock objects.

- Account may be blocked to all transactions.
- Account may be blocked to new charges, but not payments.
- Account may be blocked to all new charges, except for tuition.
- Account may allow all charges, but this transaction may fail as it takes the account over a credit limit.
- Account may allow all payments except for checks, due to the student passing a number of bad checks in the past.

If null is passed in transactionType and amount, only the account block(s) is/are returned, if there is one. (i.e. only those blocks that apply to all transactions in all cases.)

If an overrideCurrentBalance is passed, then this value is used during the credit limit check, rather than the actual current balance of the account. This permits pre-flighting of transactions even before transactions are created.

This method returns any AccountBlock objects that block the transaction, or in the case of no problems, a null response.

clearExpiredDeferments (accountId)

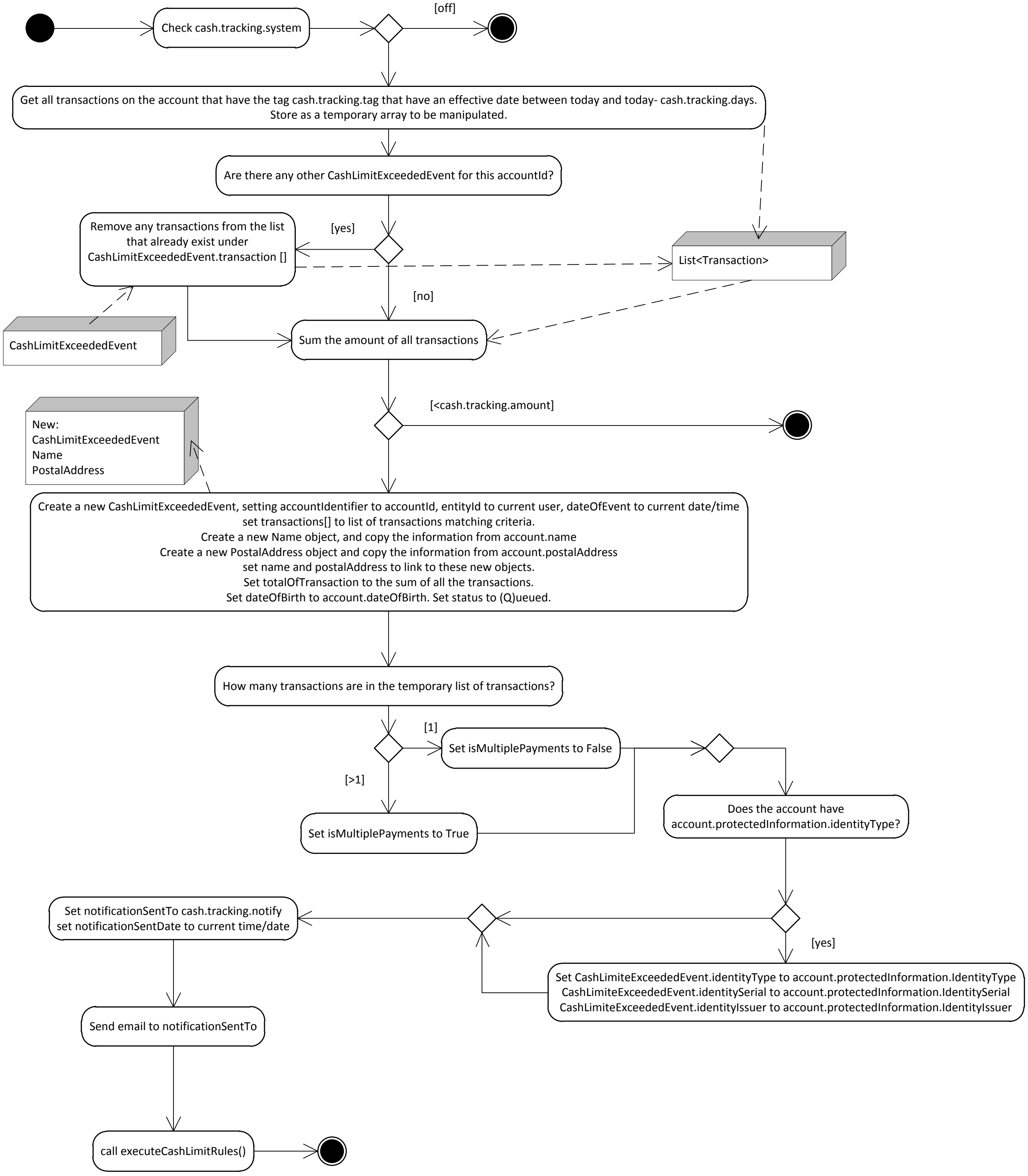
Check the expiration dates of all deferments on the account. If the date has passed, then call the expireDeferment method on each deferment.

checkCashLimit (accountId)

This process is triggered when a “cash” payment (tag is equal to cash.tracking.tag) is posted to an account. The system reviews the account for other cash payments, and if applicable, creates an event that is then passed to the cash limit rules.

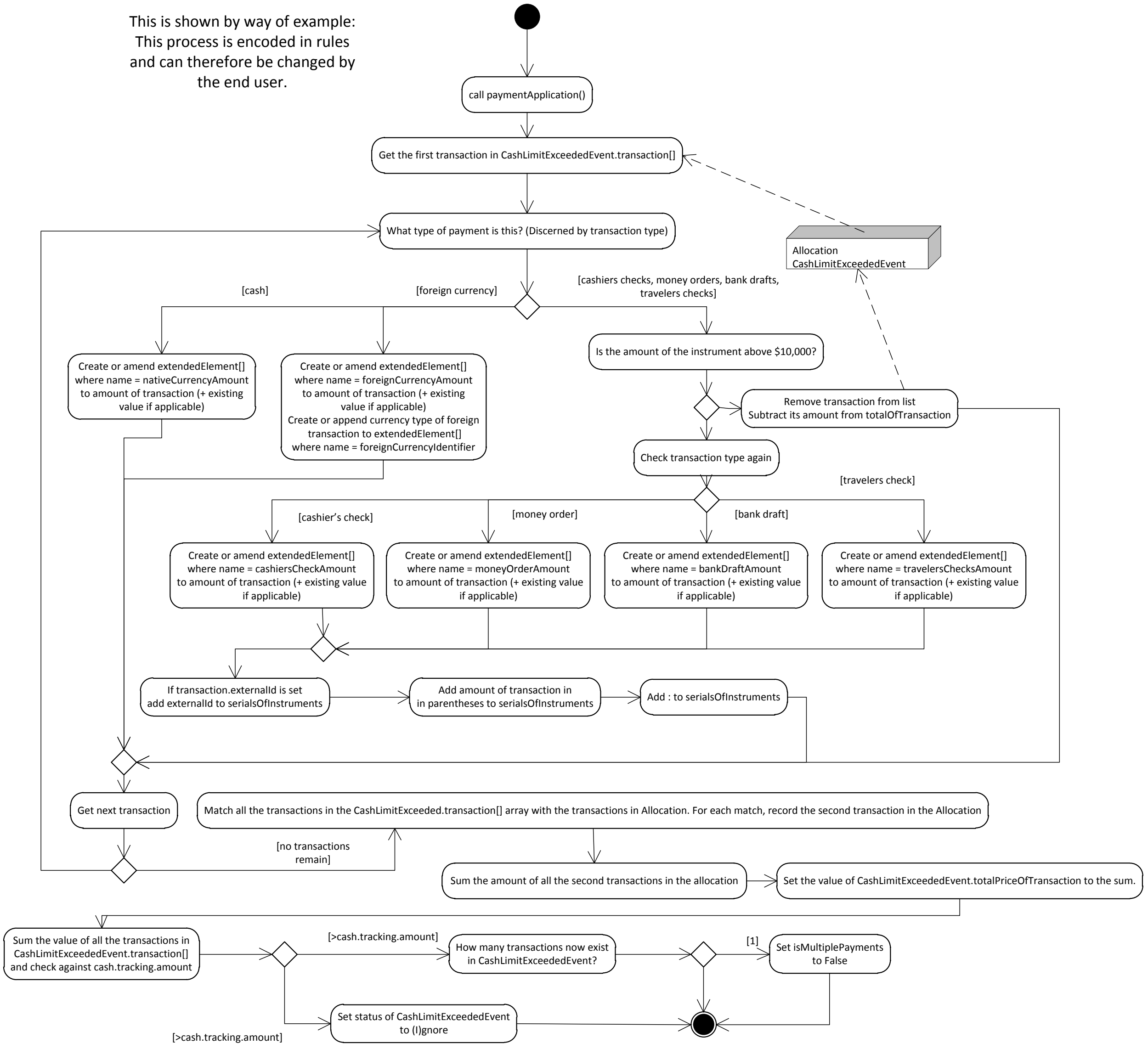
The secondary check via rules allows the institution to alter the attributes that it tracks, and apply various IRS formulae to the event before creating an 8300 (or equivalent form in other countries). This permits the system to apply rules such as certain types of transactions only count if they are below an amount (as an 8300 would have already been filed) as well as the separation of the values onto the different lines of the 8300.

The system is designed to permit collection of information related to the filing of IRS form 8300, however, it can be highly customized to permit the tracking of transactions as required for anti-laundering legislation in other jurisdictions. The amount, time period and types of transactions tracked are all configured by the end user.


Check cash.tracking.system
[off]
Get all transactions on the account that have the tag cash.tracking.tag that have an effective date between today and today- cash.tracking.days.
Store as a temporary array to be manipulated.
Are there any other CashLimitExceededEvent for this accountId?
Remove any transactions from the list that already exist under CashLimitExceededEvent.transaction []
[yes]
List<Transaction>
[no]
CashLimitExceededEvent
Sum the amount of all transactions
[<cash.tracking.amount]
New:
CashLimitExceededEvent
Name
PostalAddress
Create a new CashLimitExceededEvent, setting accountIdentifier to accountId, entityId to current user, dateOfEvent to current date/time
set transactions[] to list of transactions matching criteria.
Create a new Name object, and copy the information from account.name
Create a new PostalAddress object and copy the information from account.postalAddress
set name and postalAddress to link to these new objects.
Set totalOfTransaction to the sum of all the transactions.
Set dateOfBirth to account.dateOfBirth. Set status to (Q)ueued.
How many transactions are in the temporary list of transactions?
[1]
Set isMultiplePayments to False
[>1]
Set isMultiplePayments to True
Does the account have account.protectedInformation.identityType?
Set notificationSentTo cash.tracking.notify
set notificationSentDate to current time/date
[yes]
Set CashLimiteExceededEvent.identityType to account.protectedInformation.IdentityType
CashLimiteExceededEvent.identitySerial to account.protectedInformation.IdentitySerial
CashLimiteExceededEvent.identityIssuer to account.protectedInformation.IdentityIssuer
Send email to notificationSentTo
call executeCashLimitRules()

executeCashLimitRules (cashLimitExceededEvent)

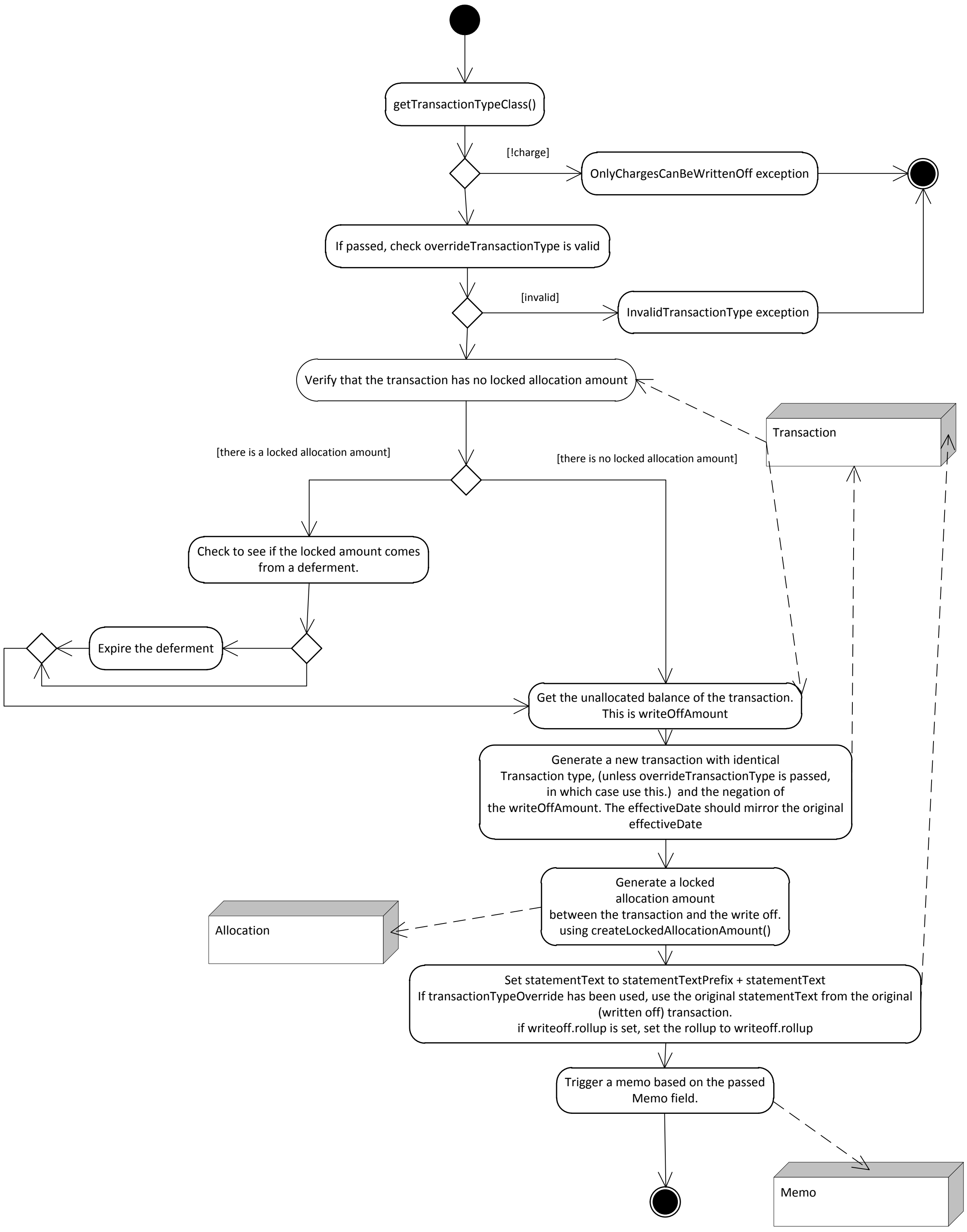
This is shown by way of example: This process is encoded in rules and can therefore be changed by the end user.


call paymentApplication()
Get the first transaction in CashLimitExceededEvent.transaction[]
What type of payment is this? (Discerned by transaction type)
Allocation
CashLimitExceededEvent
[cash]
[foreign currency]
[cashiers checks, money orders, bank drafts, travelers checks]
Create or amend extendedElement[] where name = nativeCurrencyAmount to amount of transaction (+ existing value if applicable)
Create or amend extendedElement[] where name = foreignCurrencyAmount to amount of transaction (+ existing value if applicable)
Create or append currency type of foreign transaction to extendedElement[] where name = foreignCurrencyIdentifier
Is the amount of the instrument above $10,000?
Remove transaction from list
Subtract its amount from totalOfTransaction
Check transaction type again
[travelers check]
[cashier's check]
[money order]
[bank draft]
Create or amend extendedElement[] where name = cashiersCheckAmount to amount of transaction (+ existing value if applicable)
Create or amend extendedElement[] where name = moneyOrderAmount to amount of transaction (+ existing value if applicable)
Create or amend extendedElement[] where name = bankDraftAmount to amount of transaction (+ existing value if applicable)
Create or amend extendedElement[] where name = travelersChecksAmount to amount of transaction (+ existing value if applicable)
If transaction.externalId is set add externalId to serialsOfInstruments
Add amount of transaction in in parentheses to serialsOfInstruments
Add : to serialsOfInstruments
Get next transaction
Match all the transactions in the CashLimitExceeded.transaction[] array with the transactions in Allocation. For each match, record the second transaction in the Allocation
[no transactions remain]
Sum the amount of all the second transactions in the allocation
Set the value of CashLimitExceededEvent.totalPriceOfTransaction to the sum.
Sum the value of all the transactions in CashLimitExceededEvent.transaction[] and check against cash.tracking.amount
[>cash.tracking.amount]
How many transactions now exist in CashLimitExceededEvent?
[1]
Set isMultiplePayments to False
Set status of CashLimitExceededEvent to (I)gnore
[>cash.tracking.amount]

writeOffTransaction (transactionId, memo, statementPrefix)

writeOffTransaction (transactionId, memo, statementPrefix, overrideTransactionType)

The logic of this is very similar to reverseTransaction(), except a partial write off is allowed, and only credits can be written off. Also, the institution can choose to write off charges to a different general ledger account, instead of the original, permitting the writing off to a general "bad debt" account, if they so choose. If parameter is not passed, then the write off will negate the original general ledger accounts that the transaction credited. These general ledger entries will be created during the standard makeEffective() run.


getTransactionTypeClass()
[!charge]
OnlyChargesCanBeWrittenOff exception
If passed, check overrideTransactionType is valid
[invalid]
InvalidTransactionType exception
Verify that the transaction has no locked allocation amount
Transaction
[there is a locked allocation amount]
[there is no locked allocation amount]
Check to see if the locked amount comes from a deferment.
Expire the deferment
Get the unallocated balance of the transaction. This is writeOffAmount
Generate a new transaction with identical Transaction type, (unless overrideTransactionType is passed, in which case use this.) and the negation of the writeOffAmount. The effectiveDate should mirror the original effectiveDate
Generate a locked allocation amount between the transaction and the write off. using createLockedAllocationAmount()
Allocation
Set statementText to statementTextPrefix + statementText
If transactionTypeOverride has been used, use the original statementText from the original (written off) transaction.
if writeoff.rollup is set, set the rollup to writeoff.rollup
Trigger a memo based on the passed Memo field.
Memo

writeoffAccount (accountId)

writeOffAccount (list<accountId>)

writeoffAccount (accountId, memo)

writeOffAccount (list<accountId>, memo)

This will call a set of rules that will do a number of things, including establishing whatever accounts status the school wants for written off accounts, doing a final application of payments, and then calling the writeOffTransaction() on any remaining charges. The school can also use this set of rules to perform any other events they need to on accounts that are being written off.

bounceTransaction (transactionId, memo)

Checks the transaction is a payment, and then reverses it.

System then calls a business rule to decide if a charge is to be made for the bounced transaction. Business rule is responsible for assessing the charge, should one be necessary. This logic can also set any flags as needed, such as a "Bad Check" flag, etc. transferTransaction (transactionId, amount, memo, receivingAccount, statementPrefixFromAccount, statementPrefixToAccount)

transferTransaction (transactionId, amount, memo, receivingAccount, statementPrefixFromAccount, statementPrefixToAccount)

transferTransaction (transactionId, amount, memo, receivingAccount, statementPrefixFromAccount, statementPrefixToAccount, glOverride)

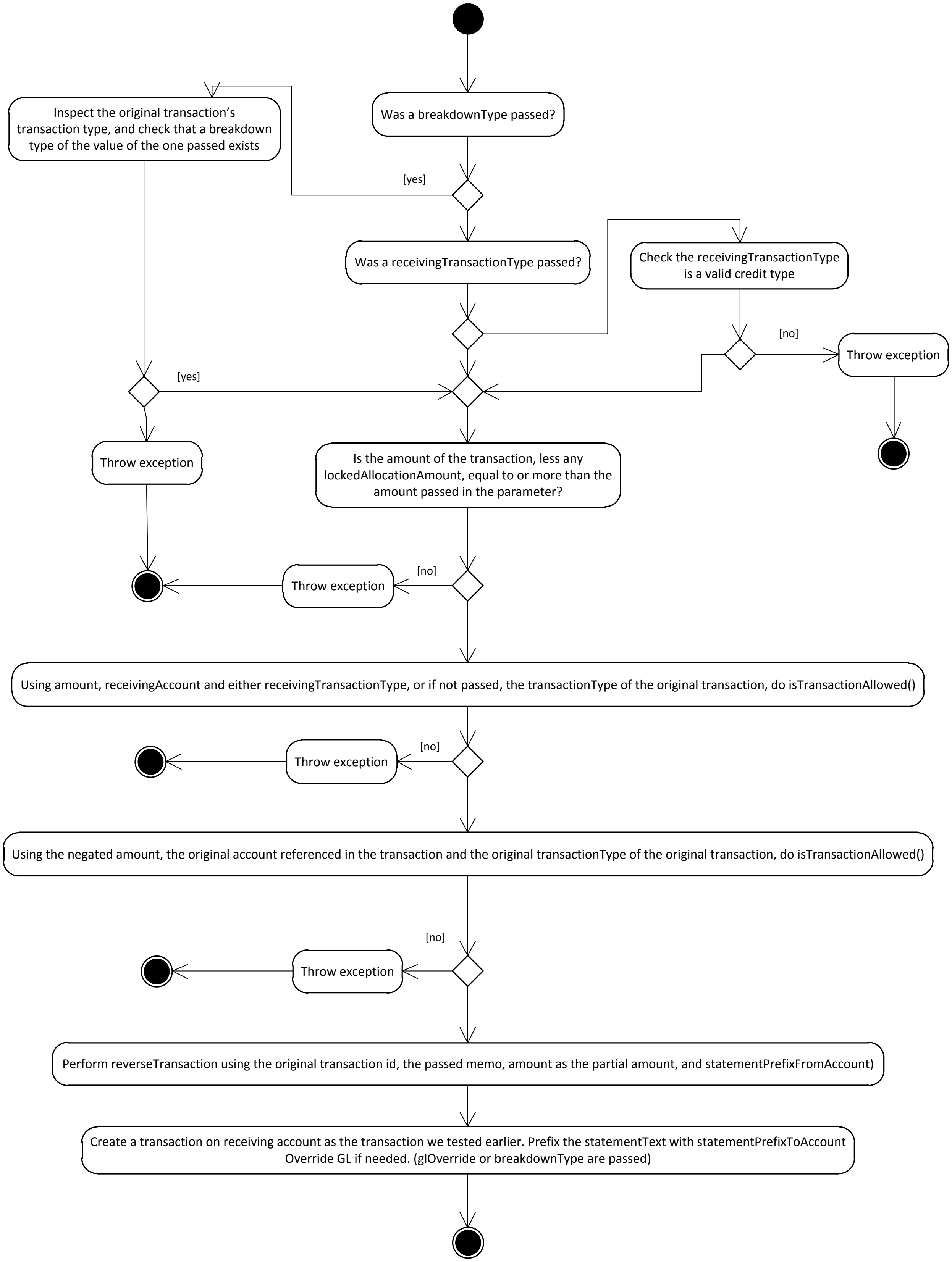
transferTransaction (transactionId, amount, memo, receivingAccount, statementPrefixFromAccount, statementPrefixToAccount, breakdownType)

transferTransaction (transactionId, amount, memo, receivingAccount, statementPrefixFromAccount, statementPrefixToAccount, receivingTransactionType)

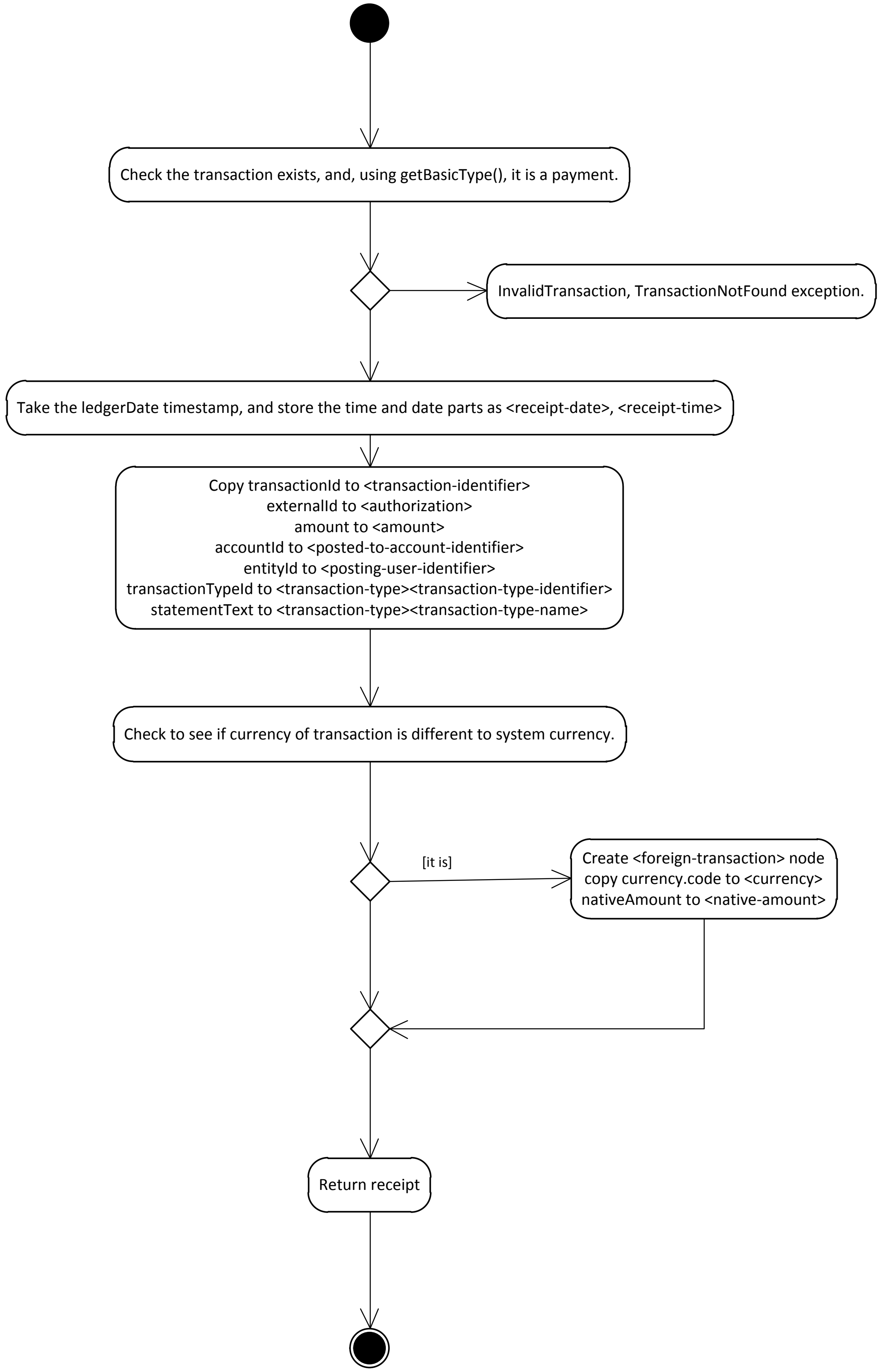
Transfer transaction is a method that transfers responsibility for a transaction from one account to another. It does this by issuing a negative transaction on the original account to wipe out its value (and its general ledger effect, if appropriate) and creates a new transaction on the new account.

In certain circumstances, the newly created transaction will need to create different general ledger transactions than the original transaction. This can be done either by passing a new glOverride block, a new breakdownType (which is related to the original transaction type) or by issuing the transaction under an entirely different transactionType.

Amount is passed, allowing only part of a transaction to be moved to a new account. For example, a sponsor may agree to pay 80% of a student's tuition charges, therefore only 80% of the tuition charge would be transferred.


Was a breakdownType passed?
[yes]
Inspect the original transaction's transaction type, and check that a breakdown type of the value of the one passed exists
Was a receivingTransactionType passed?
Check the receivingTransactionType is a valid credit type
[no]
Throw exception
[yes]
Throw exception
Is the amount of the transaction, less any lockedAllocationAmount, equal to or more than the amount passed in the parameter?
[no]
Throw exception
Using amount, receivingAccount and either receivingTransactionType, or if not passed, the transactionType of the original transaction, do isTransactionAllowed()
[no]
Throw exception
Using the negated amount, the original account referenced in the transaction and the original transactionType of the original transaction, do isTransactionAllowed()
[no]
Throw exception
Perform reverseTransaction using the original transaction id, the passed memo, amount as the partial amount, and statementPrefixFromAccount)
Create a transaction on receiving account as the transaction we tested earlier. Prefix the statementText with statementPrefixToAccount
Override GL if needed. (glOverride or breakdownType are passed)

produceReceipt (transasctionId)


Check the transaction exists, and, using getBasicType(), it is a payment.
InvalidTransaction, TransactionNotFound exception.
Take the ledgerDate timestamp, and store the time and date parts as <receipt-date>, <receipt-time>
Copy transactionId to <transaction-identifier>
externalId to <authorization>
amount to <amount>
accountId to <posted-to-account-identifier>
entityId to <posting-user-identifier>
transactionTypeId to <transaction-type><transaction-type-identifier>
statementText to <transaction-type><transaction-type-name>
Check to see if currency of transaction is different to system currency.
[it is]
Create <foreign-transaction> node
copy currency.code to <currency>
nativeAmount to <native-amount>
Return receipt

Information Service

Memos

createNewMemo (memo, level, accountIdentifier, effectiveDate, expirationDate)

createNewMemo (memo, level, accountIdentifier, effectiveDate, expirationDate, transactionId)

Instantiates a new memo on the system. Optionally a transactionId can be referenced in the memo. If there is no expirationDate of the memo, then this should be set to null.

createFollowUpMemo (memo, level, accountIdentifier, effectiveDate, expirationDate, precedingMemoIdentifier)

createFollowUpMemo (memo, level, accountIdentifier, effectiveDate, expirationDate, transactionId, precedingMemoIdentifier)

Create a memo as per createNewMemo, but first verify that the memo referenced by precedingMemoIdentifier has a null nextMemo attribute (otherwise throw an exception). Create the new memo, then set this.previousMemo to precedingMemoIdentifier, and set precedingMemoIdentifier.nextMemo to this.id

editMemo (memoId, newMemo)

Certain users are empowered to edit a memo. Using this service, the original memo text can be replaced. The editorId and lastUpdate attributes will be set during the process.

Flags

createNewFlag (flagType, level, accountIdentifier, effectiveDate, expirationDate, severity)

Instantiates a new flag (not FlagType) on the system. Severity must be greater than 0.

changeFlagSeverity (flag, newSeverity)

Certain users can change other people's flag's severity. A user can change their own flag's severity. A change will cause an update to editorId and lastUpdate. Flag severity must be greater than 0.

isFlagActive (accountIdentifier, flagType)

For a given account, is the flag type passed by flag active? Returns either a zero, for no flag set, or the flag's severity, if the flag is set.

Alerts

createNewAlert(alert, level, accountIdentifier, effectiveDate, expirationDate)

createNewAlert (alert, level, accountIdentifier, effectiveDate, expirationDate, transactionId)

Instantiates a new alert on the system. Optionally a transactionId can be referenced in the alert. If there is no expirationDate of the alert, then this should be set to null. If there is no level set in the constructor, then the level defaults to 1.

editAlert (alert, newAlertText)

Changes the text for the alert, if permitted. A change will cause an update to editorId and lastUpdate.

General

changeLevel (memoIdentifier, newLevel)

Certain users will be permitted to change the level of a piece of information. Any user can set the level to their own level or below. A change will cause an update to editorId and lastUpdate.

expire()

Pieces of information can be expired, which means that they are no longer "effective". Generally a piece of information is only displayed to a user if it is not expired. When flags are used in rules, only those that are effective are interpreted to be valid. Generally, only certain users can expire a piece of information but a user can expire a piece of information they themselves have added. A change will cause an update to editorId and lastUpdate.

linkToTransaction (information, transaction)

Link the transaction to the piece of information by saving it in the transaction attribute. This will overwrite any previous link if it was there.

setNewEffectiveDate (information, newEffectiveDate)

setNewExpirationDate (information, newExpirationDate)

Check to see that the new date is valid (information cannot expire before it is effective) and then alter the appropriate attribute.

General Ledger Service (Part of KSA-RR)

addGeneralLedgerTransaction(transactionIdentifier, generalLedgerAccount, amount, generatedInformation)

addGeneralLedgerTransaction(transactionIdentifier, generalLedgerAccount, amount, generatedInformation, isQueued)

Simple add function. Get the next glTransactionIdentifier and take the constructor details and complete the object. Set the glTransactionDate to the current date/time stamp. Set status to Q unless isQueued is passed and is false, in which case, set status to W(aiting). Set transmission to null.

createRecognitionPeriod (effectiveDateFrom, effectiveDateTo, recognitionPeriod)

createRecognitionPeriod (recognitionDateFrom, recognitionDateTo, recognitionPeriod)

Get all queued or in session general ledger transactions within the date range specified, and add the string recognitionPeriod to the transaction.

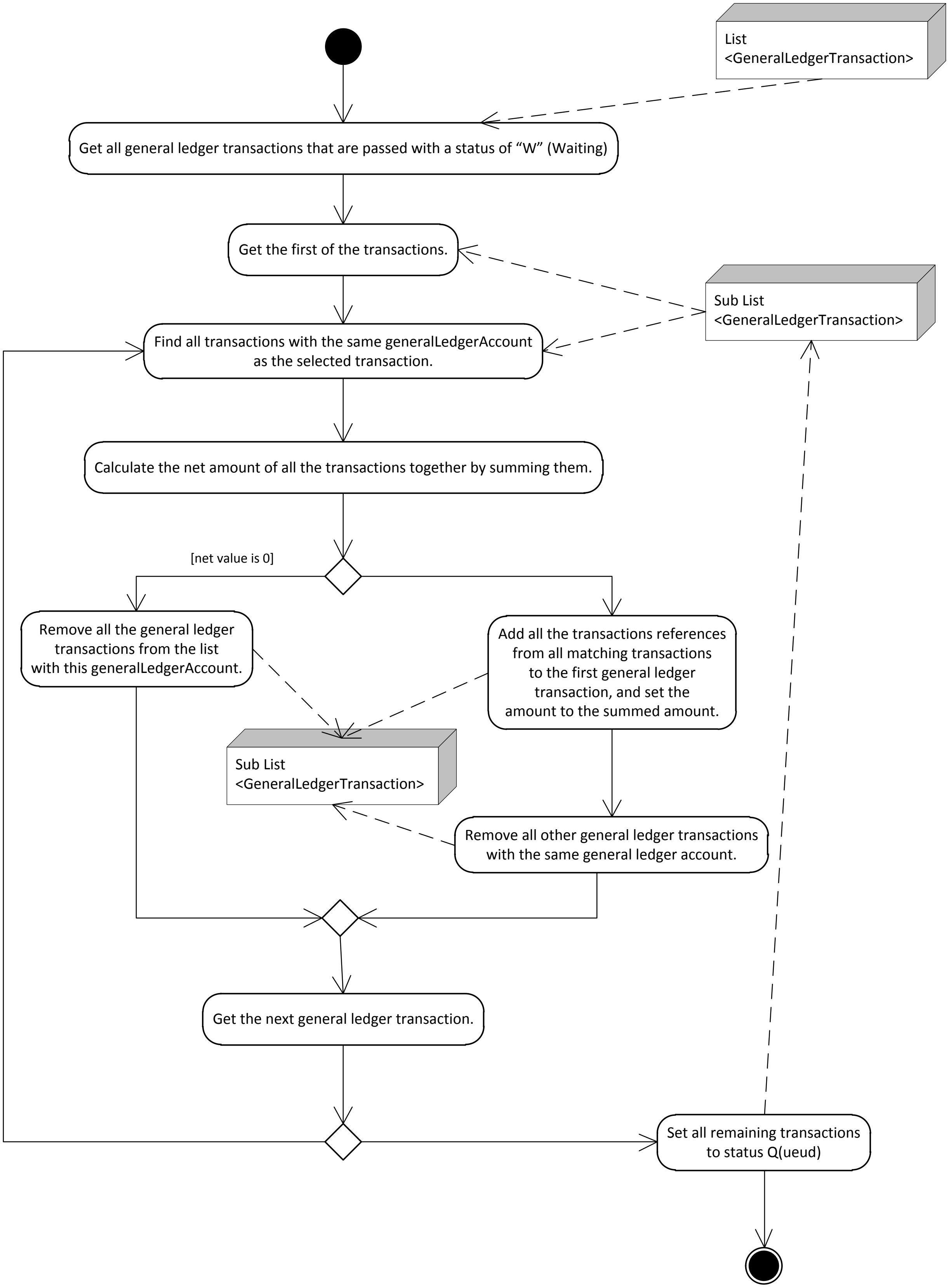
isGlAccountValid (glAccount)

This is a service that is designed to be overridden. In the standard release, this will check the account with KFS.

searchForGeneralLedgerAccounts (generalLedgerAccount)

Returns a list of transactionType where the generalLedgerAccount exists in the generalLedgerBreakdown

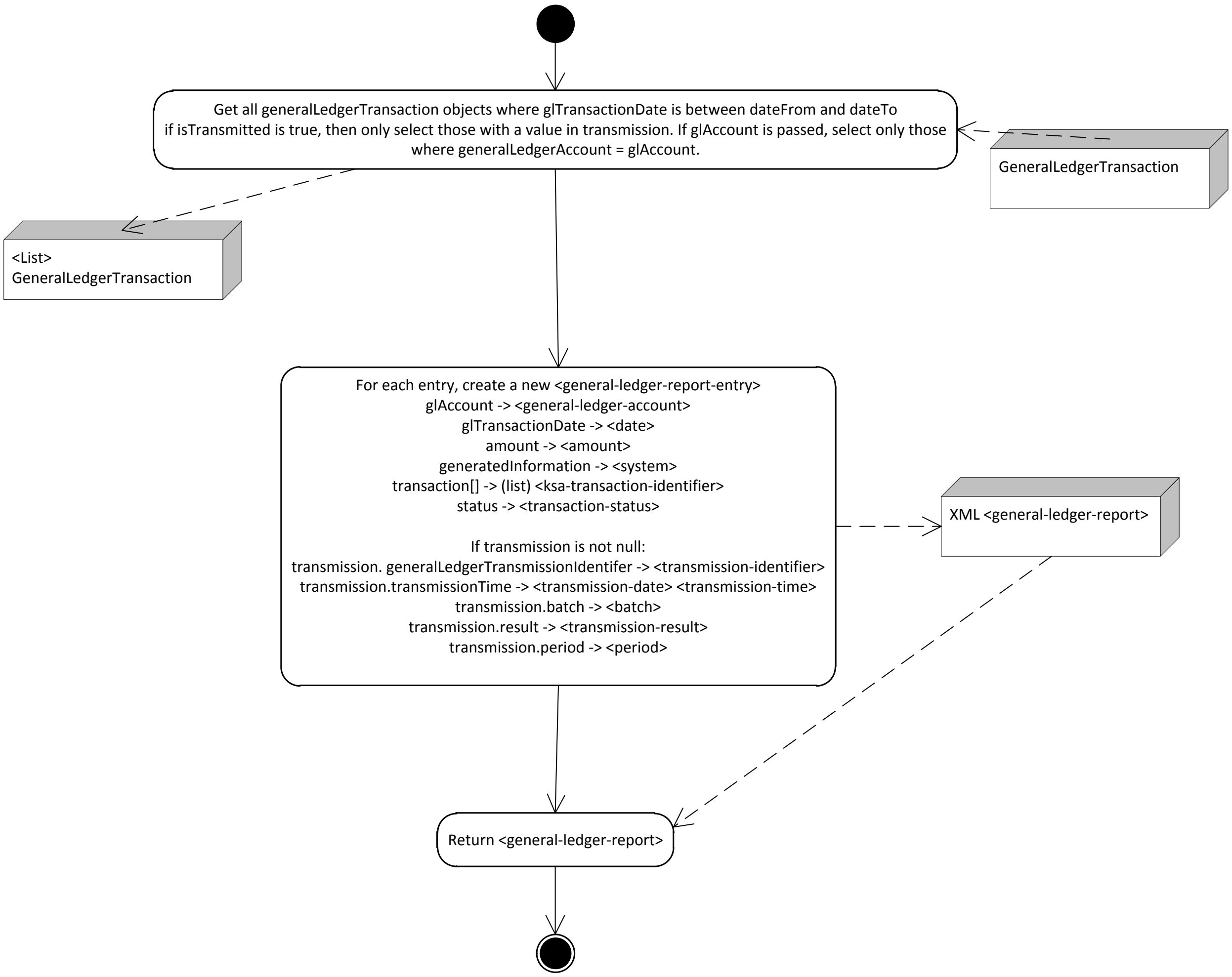
summarizeGeneralLedgerTransactions (list<generalLedgerTransaction>)


List
<GeneralLedgerTransaction>
Get all general ledger transactions that are passed with a status of “W” (Waiting)
Get the first of the transactions.
Sub List
<GeneralLedgerTransaction>
Find all transactions with the same generalLedgerAccount as the selected transaction.
Calculate the net amount of all the transactions together by summing them.
[net value is 0]
Remove all the general ledger transactions from the list with this generalLedgerAccount.
Add all the transactions references from all matching transactions to the first general ledger transaction, and set the amount to the summed amount.
Sub List
<GeneralLedgerTransaction>
Remove all other general ledger transactions with the same general ledger account.
Get the next general ledger transaction.
Set all remaining transactions to status Q(ueud)

prepareGeneralLedgerReport (dateFrom, dateTo, isTransmitted)

prepareGeneralLedgerReport (dateFrom, dateTo, isTransmitted, generalLedgerAccount)

Prepares a reconciliation report for KSA transactions to the general ledger. The schema for this report can be found in the consolidated schemata document.


Get all generalLedgerTransaction objects where glTransactionDate is between dateFrom and dateTo
if isTransmitted is true, then only select those with a value in transmission. If glAccount is passed, select only those
where generalLedgerAccount = glAccount.
GeneralLedgerTransaction
<List>
GeneralLedgerTransaction
For each entry, create a new <general-ledger-report-entry>
glAccount -> <general-ledger-account>
glTransactionDate -> <date>
amount -> <amount>
generatedInformation -> <system>
transaction[] -> (list) <ksa-transaction-identifier>
status -> <transaction-status>
If transmission is not null:
transmission. generalLedgerTransmissionIdentifer -> <transmission-identifier>
transmission.transmissionTime -> <transmission-date> <transmission-time>
transmission.batch -> <batch>
transmission.result -> <transmission-result>
transmission.period -> <period>
XML <general-ledger-report>
Return <general-ledger-report>

prepareGeneralLedgerTransmission()

prepareGeneralLedgerTransmission (effectiveDateFrom,effectiveDateTo, recognitionPeriod)

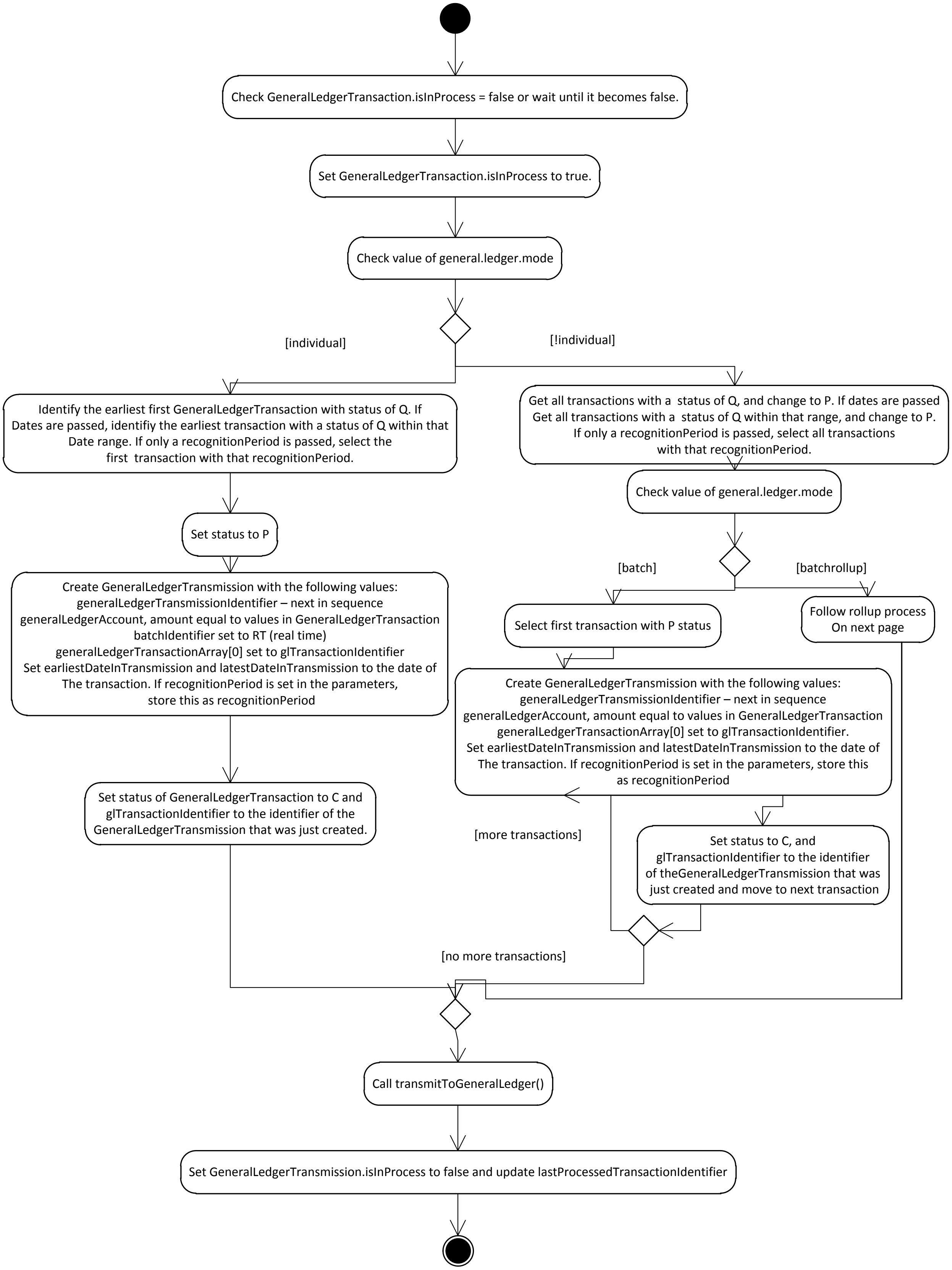
prepareGeneralLedgerTransmission (recognitionDateFrom,recognitionDateTo, recognitionPeriod)

prepareGeneralLedgerTransmission (recognitionPeriod)

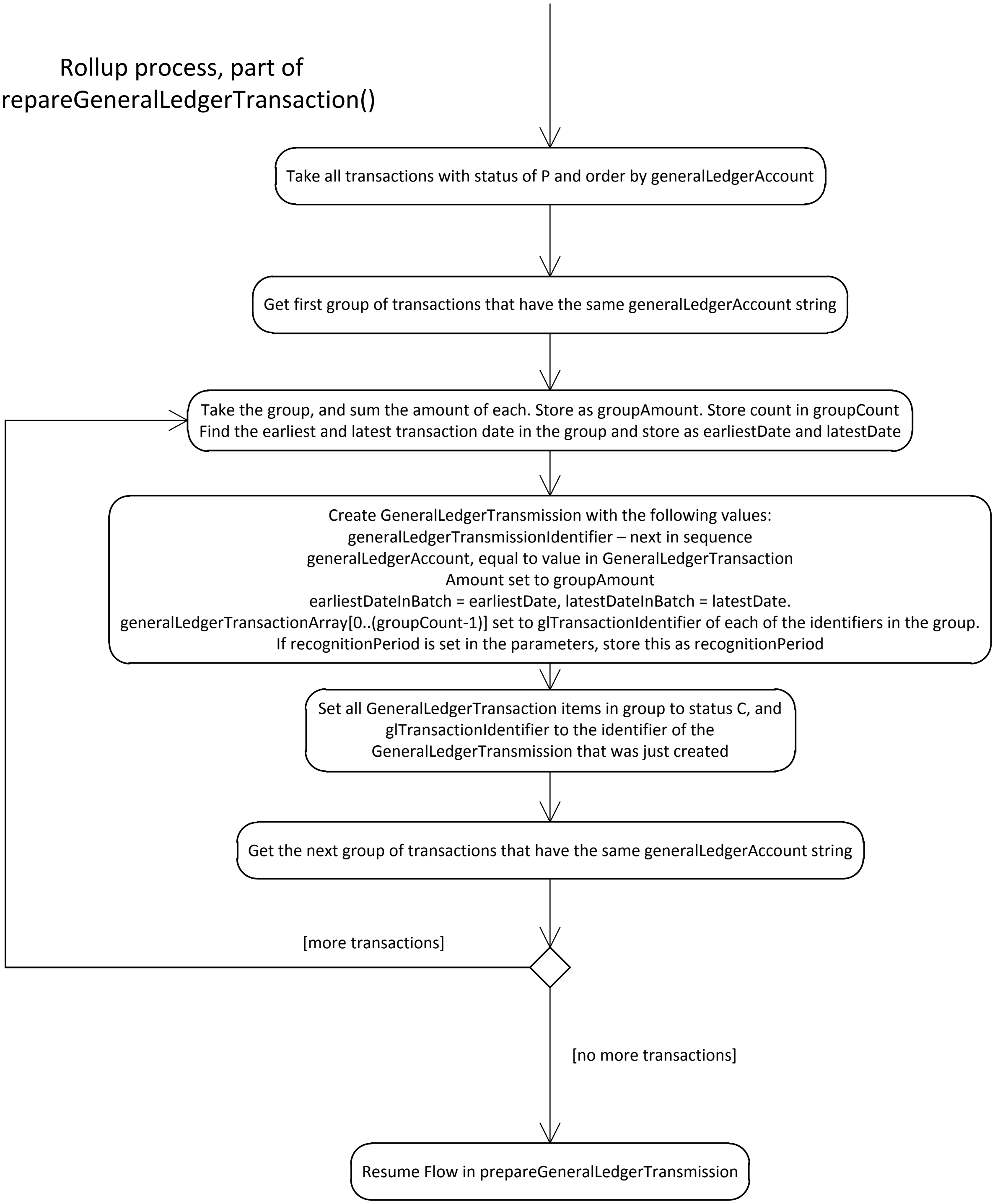
Prepares a transmission to the general ledger. This process takes into account the different ways in which an institution may choose to transmit to the general ledger, including real-time, batch, and rollup modes.

Optionally, start and end dates may be specified, along with a recognition period, allowing schools to define smaller batches to run to the general ledger with date-based recognition codes. This can also be left as null to permit date-based runs without a recognition period.

A recognitionPeriod can also be set, allowing the school to set the recognition period in advance with createRecognitionPeriod() and then send just that recognition period of transactions.


Check GeneralLedgerTransaction.isInProcess = false or wait until it becomes false.
Set GeneralLedgerTransaction.isInProcess to true.
Check value of general.ledger.mode
[individual]
[!individual]
Identify the earliest first GeneralLedgerTransaction with status of Q. If Dates are passed, identifiy the earliest transaction with a status of Q within that Date range. If only a recognitionPeriod is passed, select the first transaction with that recognitionPeriod.
Get all transactions with a status of Q, and change to P. If dates are passed Get all transactions with a status of Q within that range, and change to P. If only a recognitionPeriod is passed, select all transactions with that recognitionPeriod.
Check value of general.ledger.mode
Set status to P
[batch]
[batchrollup]
Create GeneralLedgerTransmission with the following values: generalLedgerTransmissionIdentifier – next in sequence generalLedgerAccount, amount equal to values in GeneralLedgerTransaction batchIdentifier set to RT (real time) generalLedgerTransactionArray[0] set to glTransactionIdentifier Set earliestDateInTransmission and latestDateInTransmission to the date of The transaction. If recognitionPeriod is set in the parameters, store this as recognitionPeriod
Select first transaction with P status
Follow rollup process On next page
Create GeneralLedgerTransmission with the following values: generalLedgerTransmissionIdentifier – next in sequence generalLedgerAccount, amount equal to values in GeneralLedgerTransaction generalLedgerTransactionArray[0] set to glTransactionIdentifier. Set earliestDateInTransmission and latestDateInTransmission to the date of The transaction. If recognitionPeriod is set in the parameters, store this as recognitionPeriod
Set status of GeneralLedgerTransaction to C and glTransactionIdentifier to the identifier of the GeneralLedgerTransmission that was just created.
[more transactions]
Set status to C, and glTransactionIdentifier to the identifier of theGeneralLedgerTransmission that was just created and move to next transaction
[no more transactions]
Call transmitToGeneralLedger()
Set GeneralLedgerTransmission.isInProcess to false and update lastProcessedTransactionIdentifier

Rollup process, part of prepareGeneralLedgerTransaction()


Take all transactions with status of P and order by generalLedgerAccount
Get first group of transactions that have the same generalLedgerAccount string
Take the group, and sum the amount of each. Store as groupAmount. Store count in groupCount
Find the earliest and latest transaction date in the group and store as earliestDate and latestDate
Create GeneralLedgerTransmission with the following values:
generalLedgerTransmissionIdentifier – next in sequence
generalLedgerAccount, equal to value in GeneralLedgerTransaction
Amount set to groupAmount
earliestDateInBatch = earliestDate, latestDateInBatch = latestDate.
generalLedgerTransactionArray[0..(groupCount-1)] set to glTransactionIdentifier of each of the identifiers in the group.
If recognitionPeriod is set in the parameters, store this as recognitionPeriod
Set all GeneralLedgerTransaction items in group to status C, and
glTransactionIdentifier to the identifier of the
GeneralLedgerTransmission that was just created
Get the next group of transactions that have the same generalLedgerAccount string
[more transactions]
[no more transactions]
Resume Flow in prepareGeneralLedgerTransmission

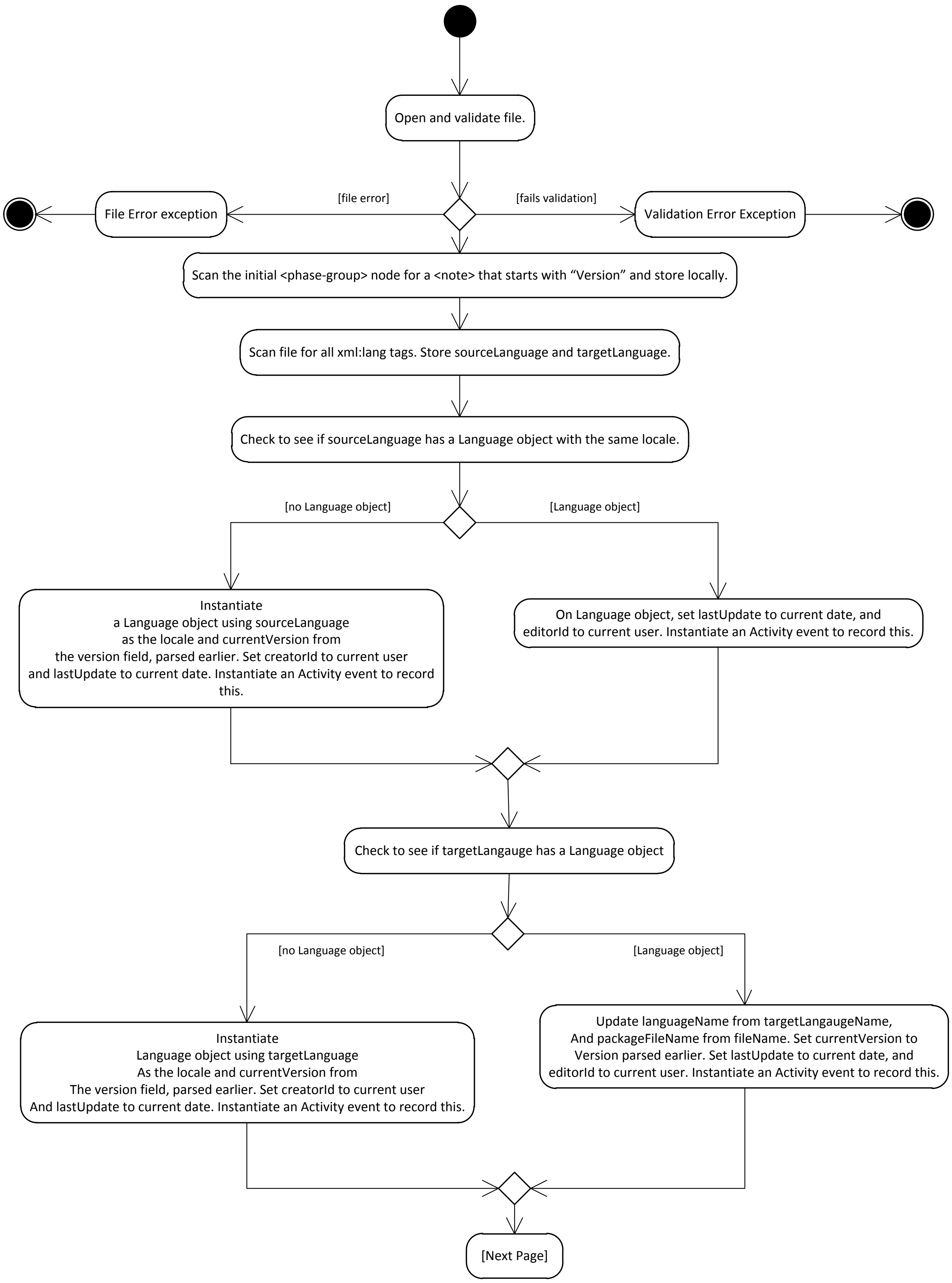
Localization Service

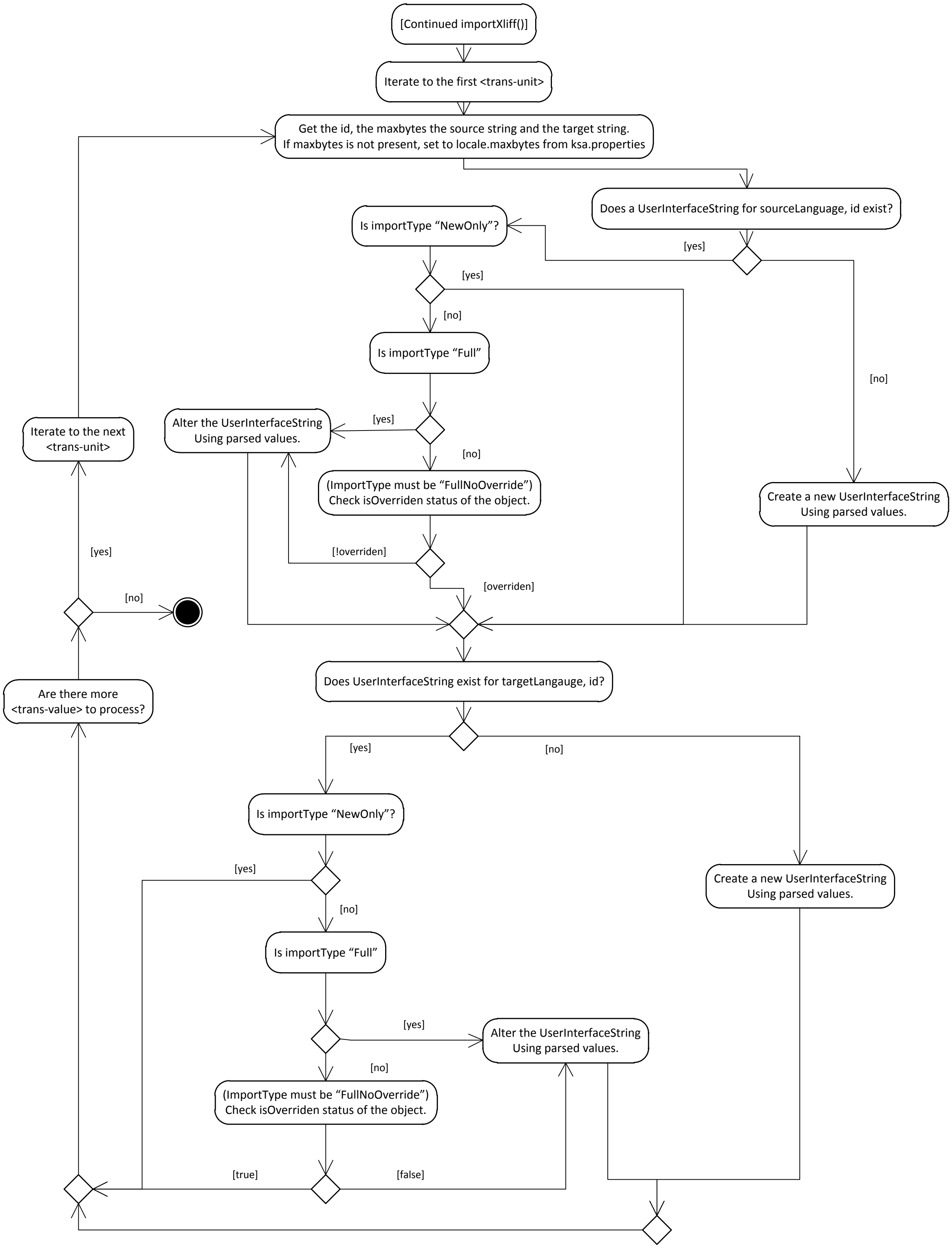
The localization service is used to import XLIFF files into the InstalledLanguage and UserInterfaceString objects to permit localization of the language of the user interface for KSA.

importResources (filename, importType [Full, FullNoOverride, NewOnly])

Imports the file located at filename.

ImportType can be set to Full (all target strings are imported no matter what), FullNoOverride, where all target strings, except for those with the isOverriden flag are imported, or NewOnly, where only target strings that have not already been imported are brought into the system. This allows the importation of a new language pack, without destroying customizations made by the institution.


Open and validate file.
[file error]
[fails validation]
File Error exception
Validation Error Exception
Scan the initial <phase-group> node for a <note> that starts with “Version” and store locally.
Scan file for all xml:lang tags. Store sourceLanguage and targetLanguage.
Check to see if sourceLanguage has a Language object with the same locale.
[no Language object]
[Language object]
Instantiate a Language object using sourceLanguage as the locale and currentVersion from the version field, parsed earlier. Set creatorId to current user and lastUpdate to current date. Instantiate an Activity event to record this.
On Language object, set lastUpdate to current date, and editorId to current user. Instantiate an Activity event to record this.
Check to see if targetLangauge has a Language object
[no Language object]
[Language object]
Instantiate Language object using targetLanguage As the locale and currentVersion from The version field, parsed earlier. Set creatorId to current user And lastUpdate to current date. Instantiate an Activity event to record this.
Update languageName from targetLangaugeName, And packageFileName from fileName. Set currentVersion to Version parsed earlier. Set lastUpdate to current date, and editorId to current user. Instantiate an Activity event to record this.
[Next Page]


[Continued importXliff()]
Iterate to the first <trans-unit>
Get the id, the maxbytes the source string and the target string.
If maxbytes is not present, set to locale.maxbytes from ksa.properties
Does a UserInterfaceString for sourceLanguage, id exist?
[yes]
[no]
Is importType "NewOnly"?
[yes]
[no]
Is importType "Full"
[yes]
[no]
Alter the UserInterfaceString
Using parsed values.
(ImportType must be "FullNoOverride")
Check isOverriden status of the object.
[!overriden]
[overriden]
Create a new UserInterfaceString
Using parsed values.
Iterate to the next
<trans-unit>
[yes]
[no]
Are there more
<trans-value> to process?
Does UserInterfaceString exist for targetLangauge, id?
[yes]
[no]
Is importType "NewOnly"?
[yes]
[no]
Is importType "Full"
[yes]
[no]
Alter the UserInterfaceString
Using parsed values.
(ImportType must be "FullNoOverride")
Check isOverriden status of the object.
[true]
[false]
Create a new UserInterfaceString
Using parsed values.

setLanguageName (languageCode, languageName)

getLanguageName (languageCode)

Refund Service (In Progress)

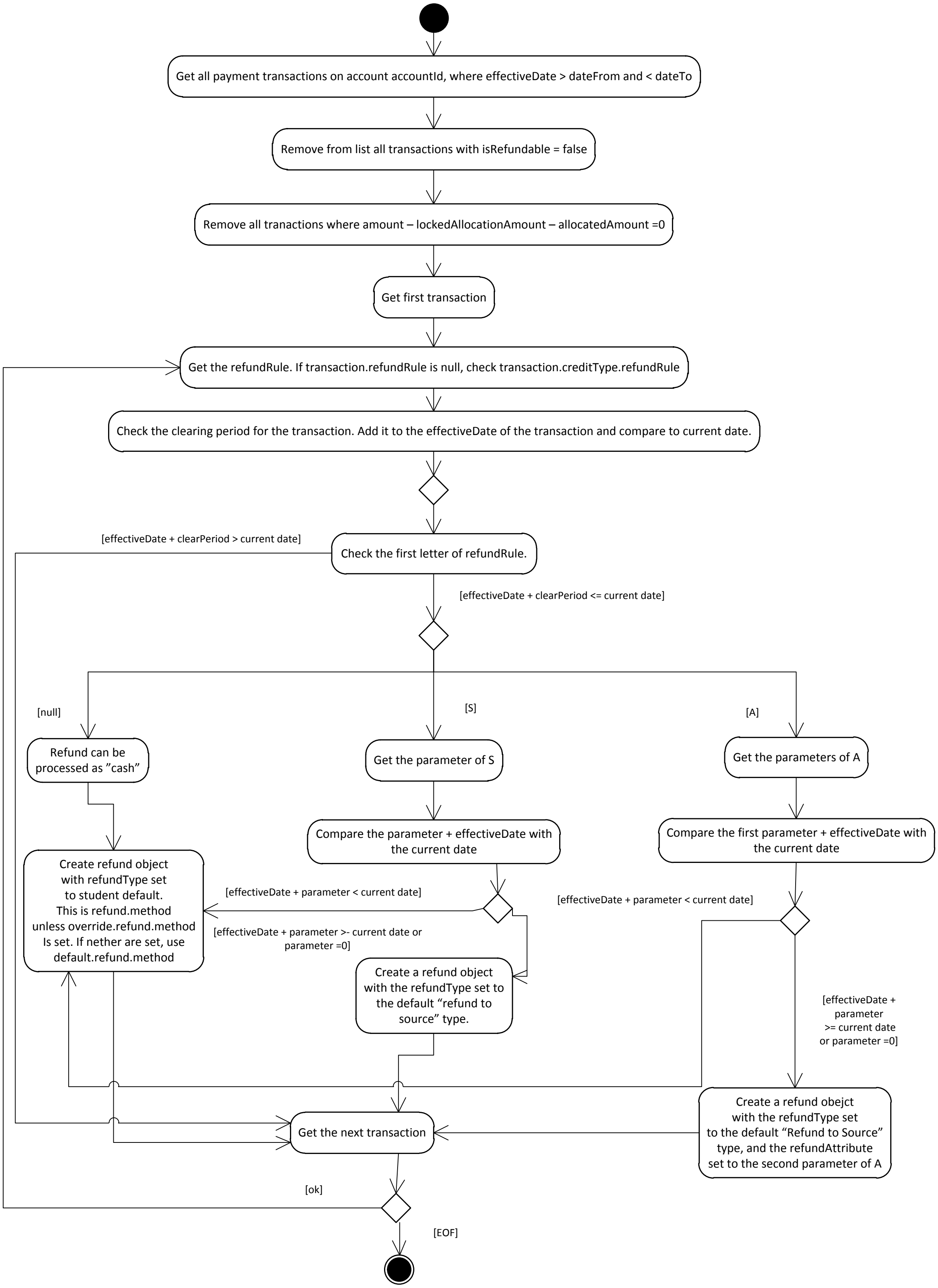
The refund service is the core of KSA-RM-RM. It handles the production of refunds from the system. At its heart is the Refund object, which acts as a queue for refunds to be processed.

The refund service will try to determine if a transaction can be paid as “cash” (that is, without restrictions, not going to a third party, or being repaid in a specific way.) If a refund (or part of a refund) can be paid as “cash” then it will be paid with the refund type as defined in refund.method, which is a student-specific attribute, unless override.refund.method is set to a different refund type, in which case, this takes precedence. If neither are set, then default.refund.method is used.

This allows a default method for all students (likely paper check) with students able to sign up for other refund types (bank transfer, etc.) and it allows a counselor to override that choice in certain cases (for example, they may require certain students to pick up their refund check at the office.)

checkForRefund (fromDate, toDate, accountId)

This creates a list of unverified refunds.


Get all payment transactions on account accountId, where effectiveDate > dateFrom and < dateTo
Remove from list all transactions with isRefundable = false
Remove all tranactions where amount – lockedAllocationAmount – allocatedAmount =0
Get first transaction
Get the refundRule. If transaction.refundRule is null, check transaction.creditType.refundRule
Check the clearing period for the transaction. Add it to the effectiveDate of the transaction and compare to current date.
[effectiveDate + clearPeriod > current date]
Check the first letter of refundRule.
[effectiveDate + clearPeriod <= current date]
[null]
[S]
[A]
Refund can be processed as "cash"
Get the parameter of S
Get the parameters of A
Compare the parameter + effectiveDate with the current date
Compare the first parameter + effectiveDate with the current date
Create refund object with refundType set to student default. This is refund.method unless override.refund.method Is set. If nether are set, use default.refund.method
[effectiveDate + parameter < current date]
[effectiveDate + parameter >- current date or parameter =0]
[effectiveDate + parameter < current date]
Create a refund object with the refundType set to the default "refund to source" type.
[effectiveDate + parameter >= current date or parameter =0]
Get the next transaction
Create a refund obejct with the refundType set to the default "Refund to Source" type, and the refundAttribute set to the second parameter of A
[ok]
[EOF]

Notes:

The syntax of “refundRule” is discussed in the data model documentation under the heading “Refund Rule Use Cases”.

paymentApplication() should be run before calling the refund routine.

It is expected that once the system has produced this refund list, an institution-specific set of rules would then check for use cases specific to the school in question and ensure that the list is ready for human validation. As an example, for an account refund (a refund from one KSA account to another KSA account) the refundAttribute can be used to override the statement text of the refund. The rules engine could be used to insert this text, based on institution-specific preferences.

checkForRefund (fromDate, toDate, list<accountId>)

For each accountId, perform a checkForRefund()

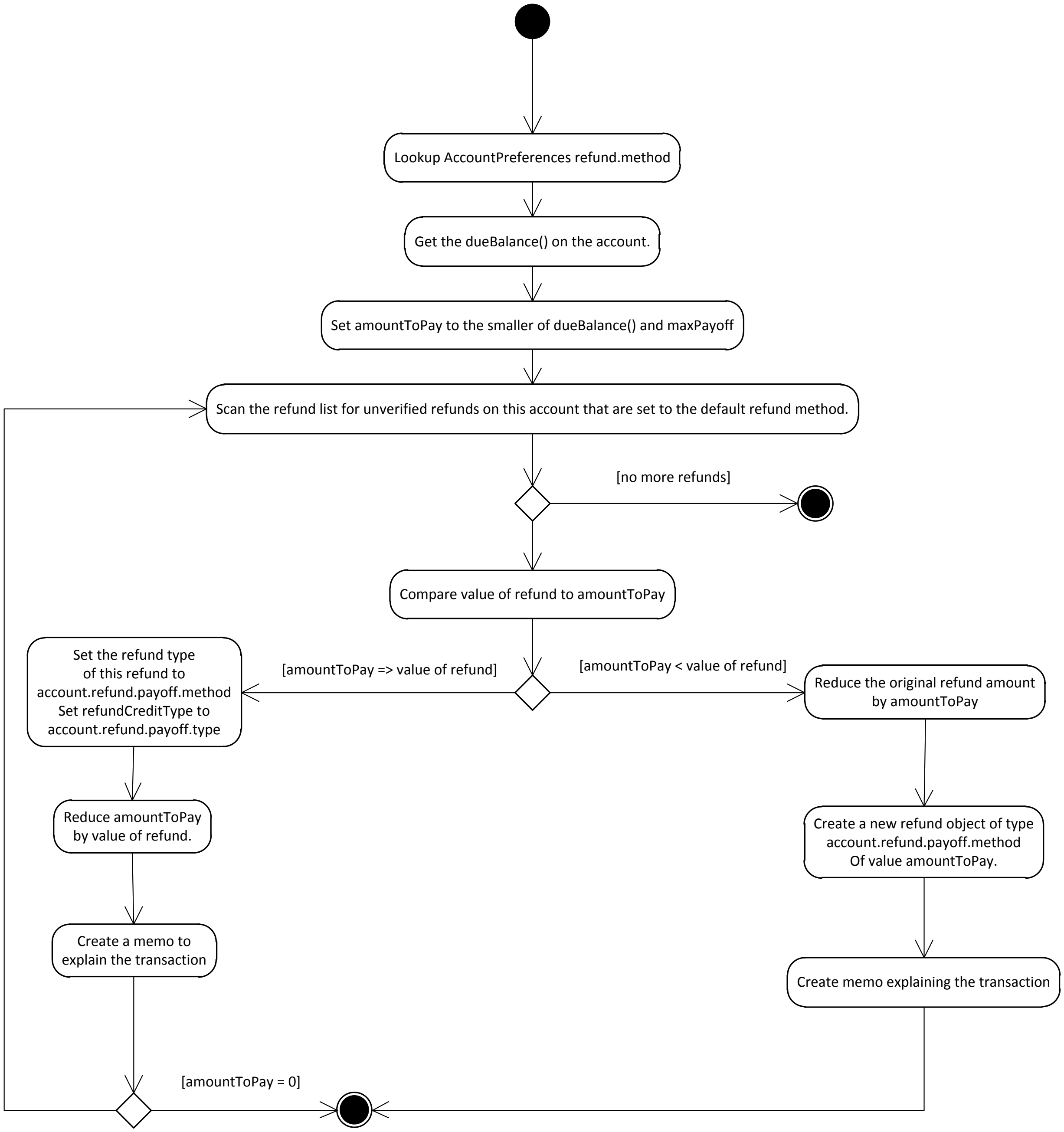
produceRefundList ()

produceRefundList (dateFrom, dateTo)

Iterates through all accounts looking for refundable amounts. For each account, checkForRefund() is called. If the dates are passed, these are passed to the checkForRefund method. If not, then maximum dates are passed.

In many cases, “produceRefundList()” will be used infrequently, as many schools will elect to run refunds against specific populations of students.

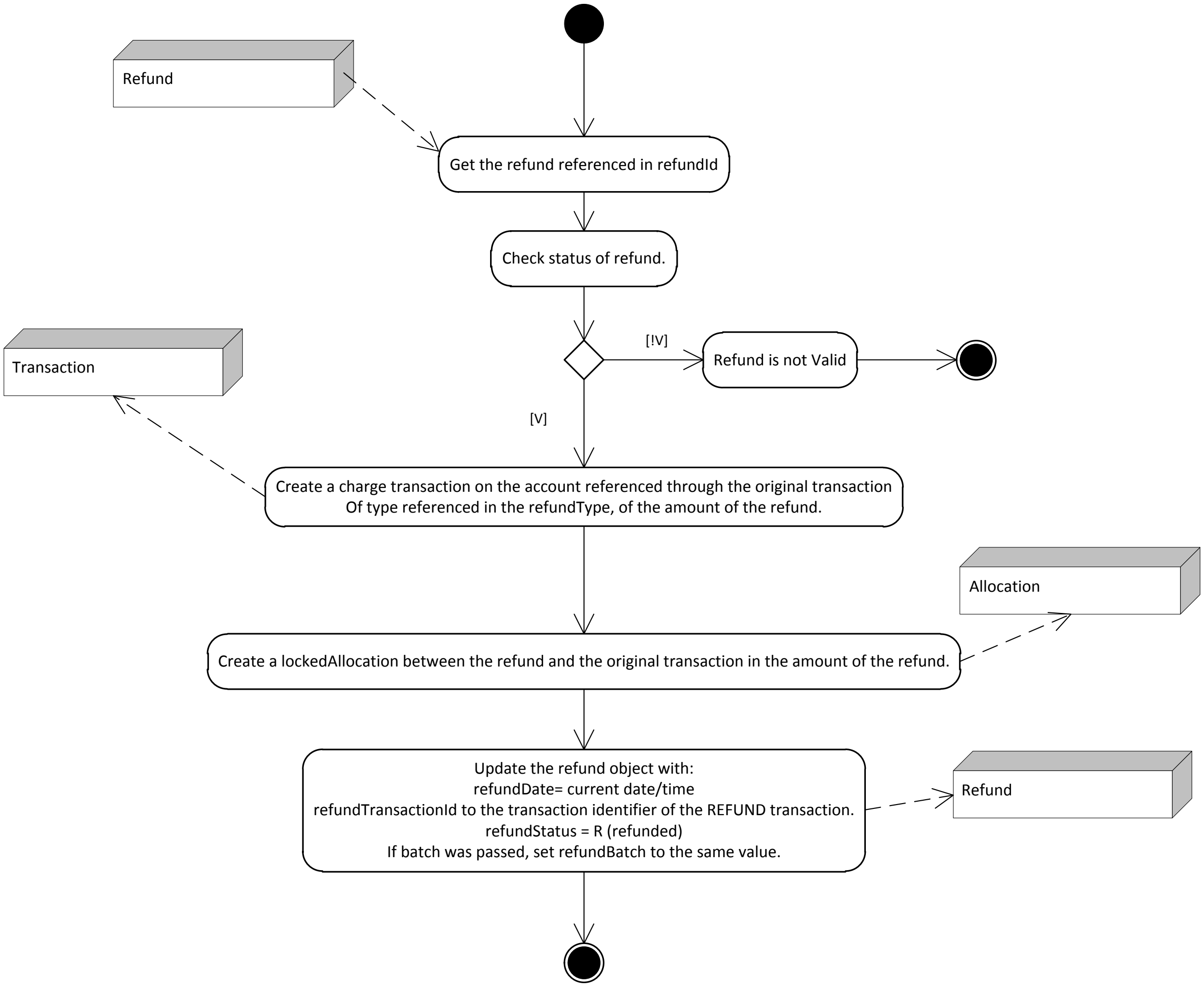
payoffWithRefund(accountId, maxPayoff)


Lookup AccountPreferences refund.method
Get the dueBalance() on the account.
Set amountToPay to the smaller of dueBalance() and maxPayoff
Scan the refund list for unverified refunds on this account that are set to the default refund method.
[no more refunds]
Compare value of refund to amountToPay
[amountToPay => value of refund]
[amountToPay < value of refund]
Set the refund type of this refund to account.refund.payoff.method Set refundCreditType to account.refund.payoff.type
Reduce the original refund amount by amountToPay
Reduce amountToPay by value of refund.
Create a new refund object of type account.refund.payoff.method Of value amountToPay.
Create a memo to explain the transaction
Create memo explaining the transaction
[amountToPay = 0]

performRefund(refundId)

performRefund(refundId, batch)

This actually creates the refund transaction, in addition to allocating the refund to the original charge. It marks the refund object as refunded.

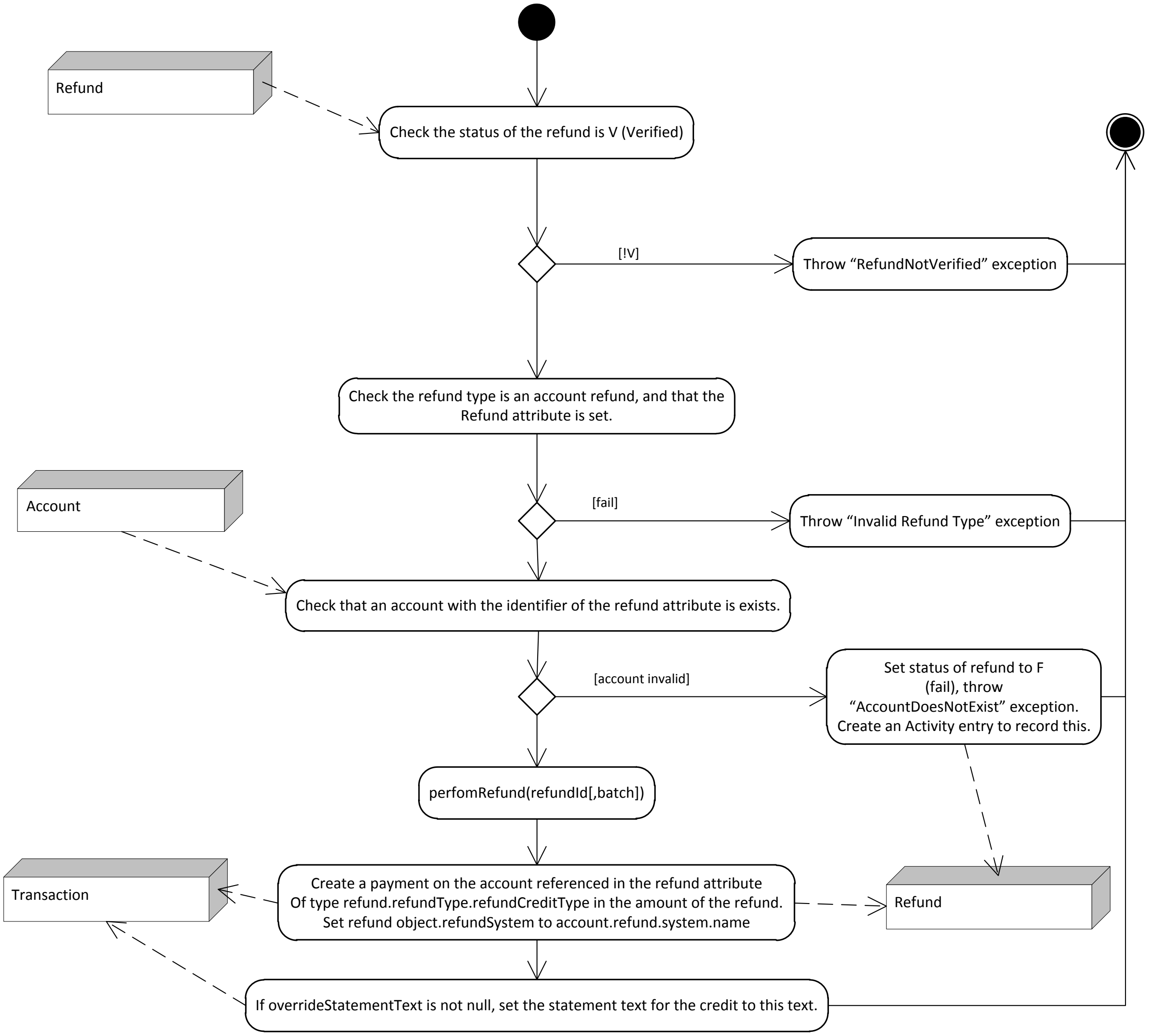

Refund
Get the refund referenced in refundId
Check status of refund.
[!V]
Refund is not Valid
Transaction
[V]
Create a charge transaction on the account referenced through the original transaction
Of type referenced in the refundType, of the amount of the refund.
Allocation
Create a lockedAllocation between the refund and the original transaction in the amount of the refund.
Update the refund object with:
refundDate= current date/time
refundTransactionId to the transaction identifier of the REFUND transaction.
refundStatus = R (refunded)
If batch was passed, set refundBatch to the same value.
Refund

validateRefund (refundId)

Sets the refundStatus to “V” and the authorizedBy to the current user.

doAccountRefund (refundId)

doAccountRefund (refundId, batch)


Refund
Check the status of the refund is V (Verified)
[!V]
Throw “RefundNotVerified” exception
Check the refund type is an account refund, and that the Refund attribute is set.
Account
[fail]
Throw “Invalid Refund Type” exception
Check that an account with the identifier of the refund attribute is exists.
[account invalid]
Set status of refund to F (fail), throw “AccountDoesNotExist” exception. Create an Activity entry to record this.
perfomRefund(refundId[,batch])
Transaction
Create a payment on the account referenced in the refund attribute Of type refund.refundType.refundCreditType in the amount of the refund. Set refund object.refundSystem to account.refund.system.name
Refund
If overrideStatementText is not null, set the statement text for the credit to this text.

doAllAccountRefunds (batch)

Go through the Refund objects. For each validated refund with type set to account refund (account.refund.type). For each one that is found, call doAccountRefund (refundId, batch).

doCheckRefund(refundId)

doCheckRefund(refundId, batch)

doCheckRefund(refundId, checkDate)

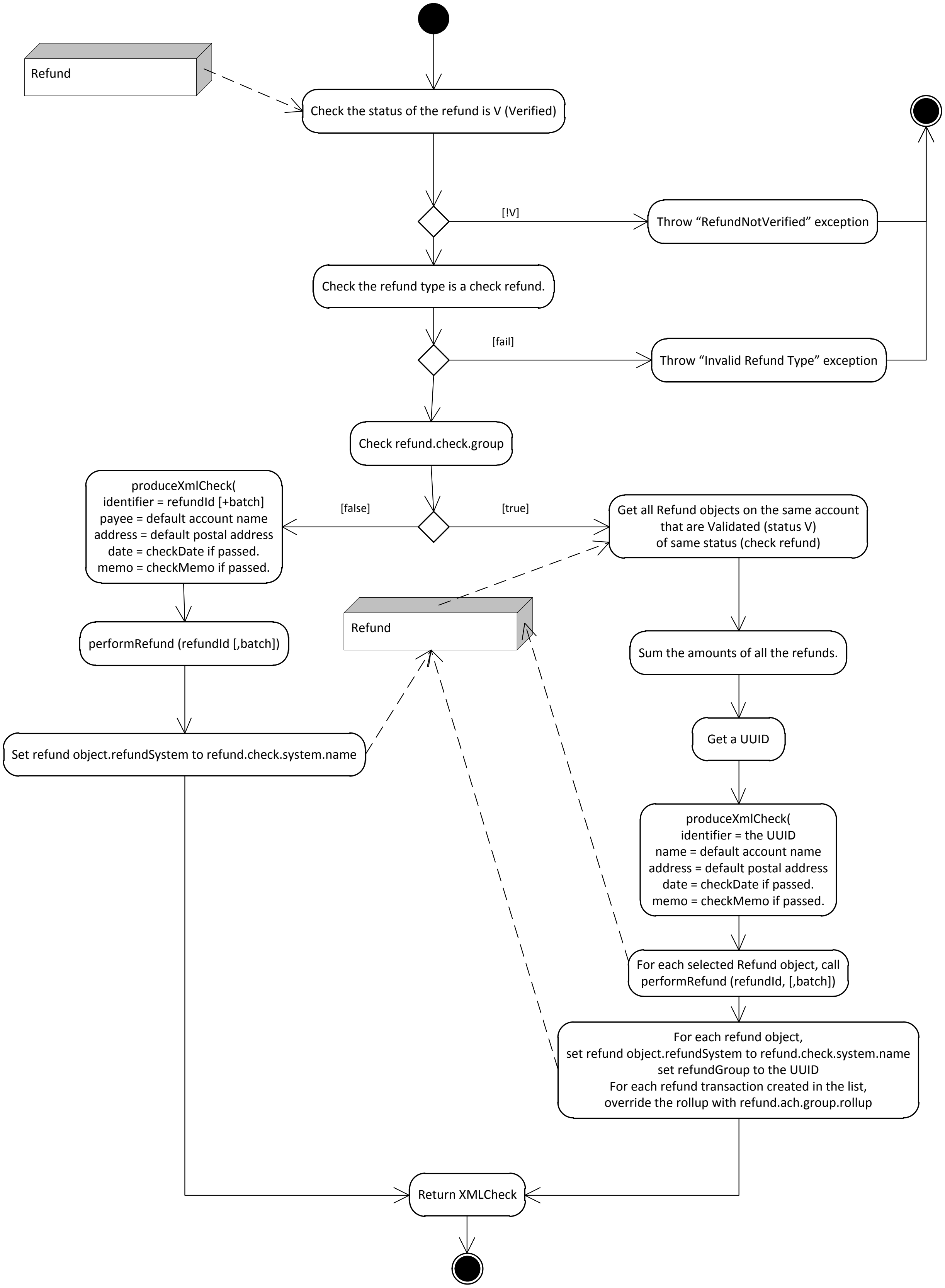
doCheckRefund(refundId, checkMemo)

doCheckRefund(refundId, batch, checkDate)

doCheckRefund(refundId, batch, checkMemo)

doCheckRefund(refundId, checkDate, checkMemo)

doCheckRefund(refundId, batch, checkDate, checkMemo)

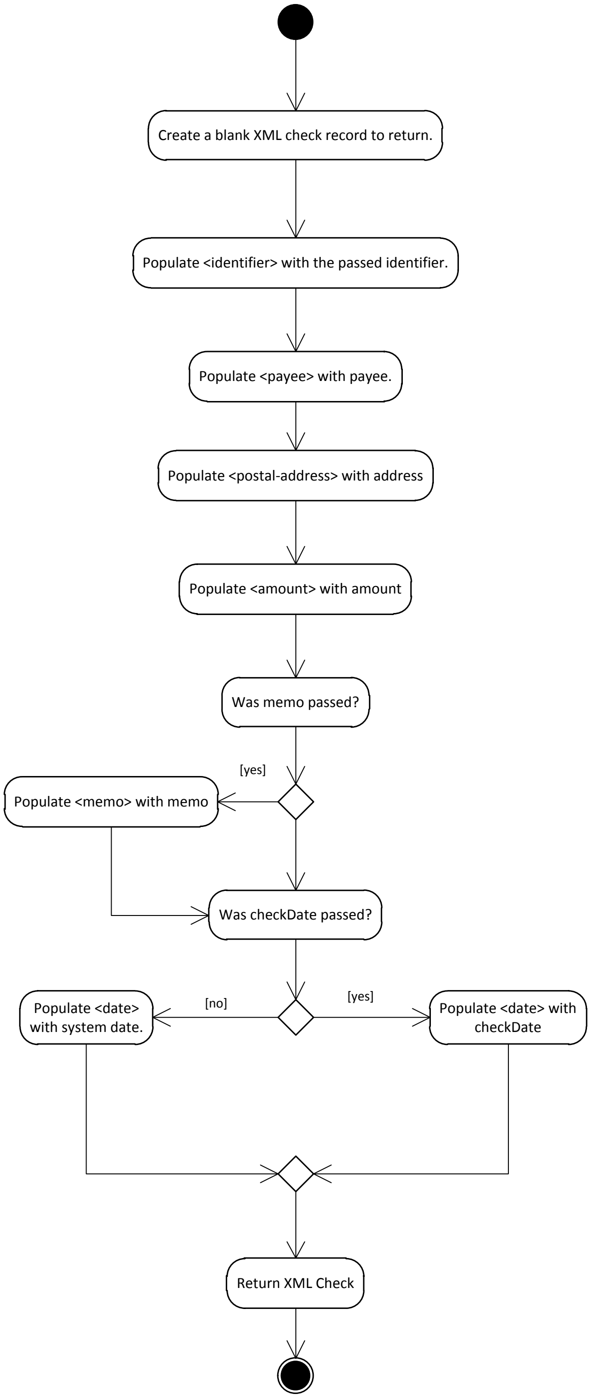

Refund
Check the status of the refund is V (Verified)
[!V]
Throw “RefundNotVerified” exception
Check the refund type is a check refund.
[fail]
Throw “Invalid Refund Type” exception
Check refund.check.group
[false]
[true]
produceXmlCheck(
identifier = refundId [+batch]
payee = default account name
address = default postal address
date = checkDate if passed.
memo = checkMemo if passed.
Get all Refund objects on the same account
that are Validated (status V)
of same status (check refund)
Refund
performRefund (refundId [,batch])
Sum the amounts of all the refunds.
Set refund object.refundSystem to refund.check.system.name
Get a UUID
produceXmlCheck(
identifier = the UUID
name = default account name
address = default postal address
date = checkDate if passed.
memo = checkMemo if passed.
For each selected Refund object, call
performRefund (refundId, [,batch])
For each refund object,
set refund object.refundSystem to refund.check.system.name
set refundGroup to the UUID
For each refund transaction created in the list,
override the rollup with refund.ach.group.rollup
Return XMLCheck

produceXmlCheck (identifier, payee, address, amount)

produceXmlCheck (identifier, payee, address, amount, memo)

produceXmlCheck (identifier, payee, address, amount, checkDate)

produceXmlCheck (identifier, payee, address, amount, memo, checkDate)


Create a blank XML check record to return.
Populate <identifier> with the passed identifier.
Populate <payee> with payee.
Populate <postal-address> with address
Populate <amount> with amount
Was memo passed?
[yes]
Populate <memo> with memo
Was checkDate passed?
[no]
[yes]
Populate <date> with system date.
Populate <date> with checkDate
Return XML Check

doAllCheckRefunds(batch)

doAllCheckRefunds(batch, checkMemo)

doAllCheckRefunds(batch, checkDate)

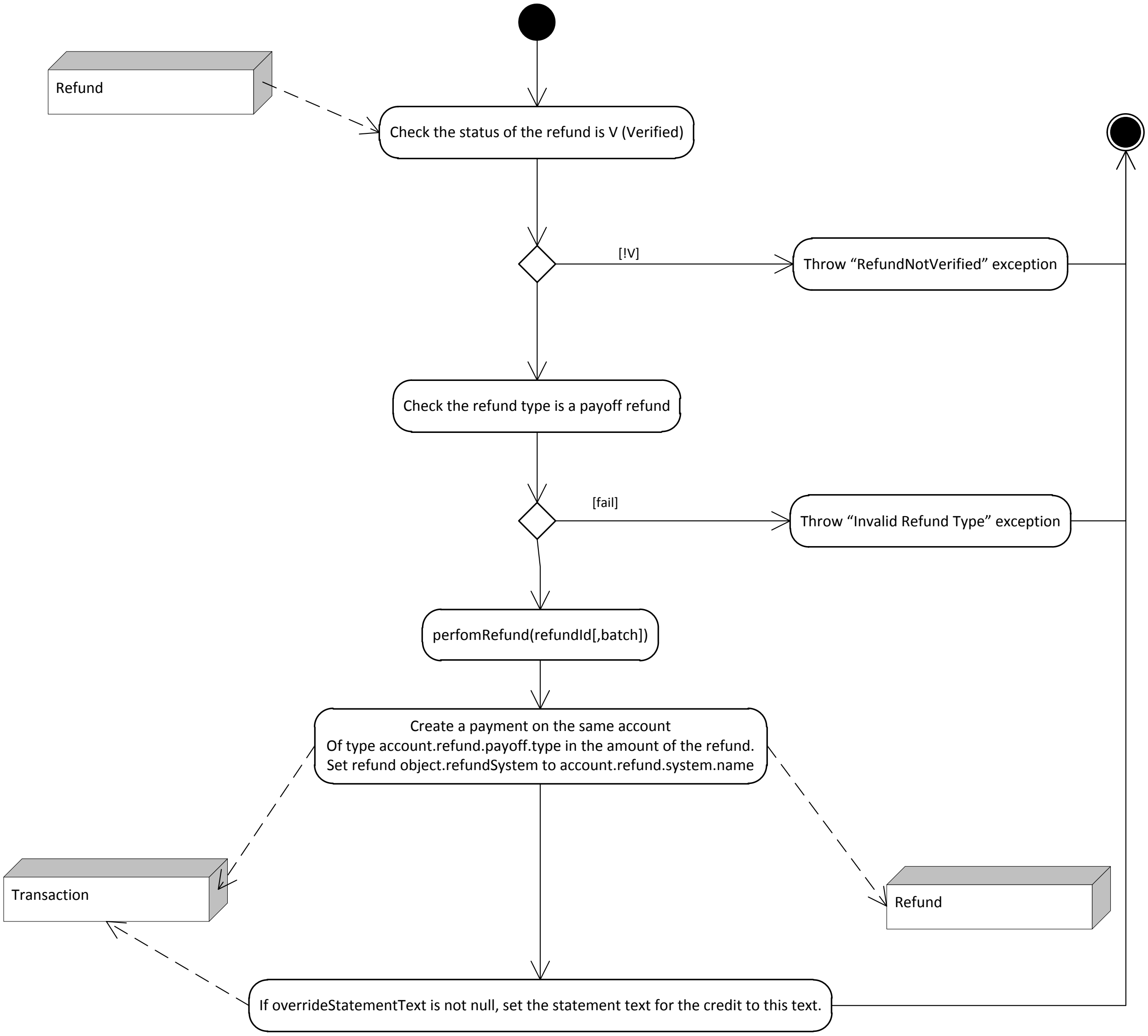
doAllCheckRefunds (batch, checkMemo)

doAllCheckRefunds(batch, checkMemo, checkDate)

Iterate through all check refunds (refund.check.type) that are validated, and produce document of <batch-check>. Return XML document.

doPayoffRefund (refundId)

doPayoffRefund (refundId, batch)

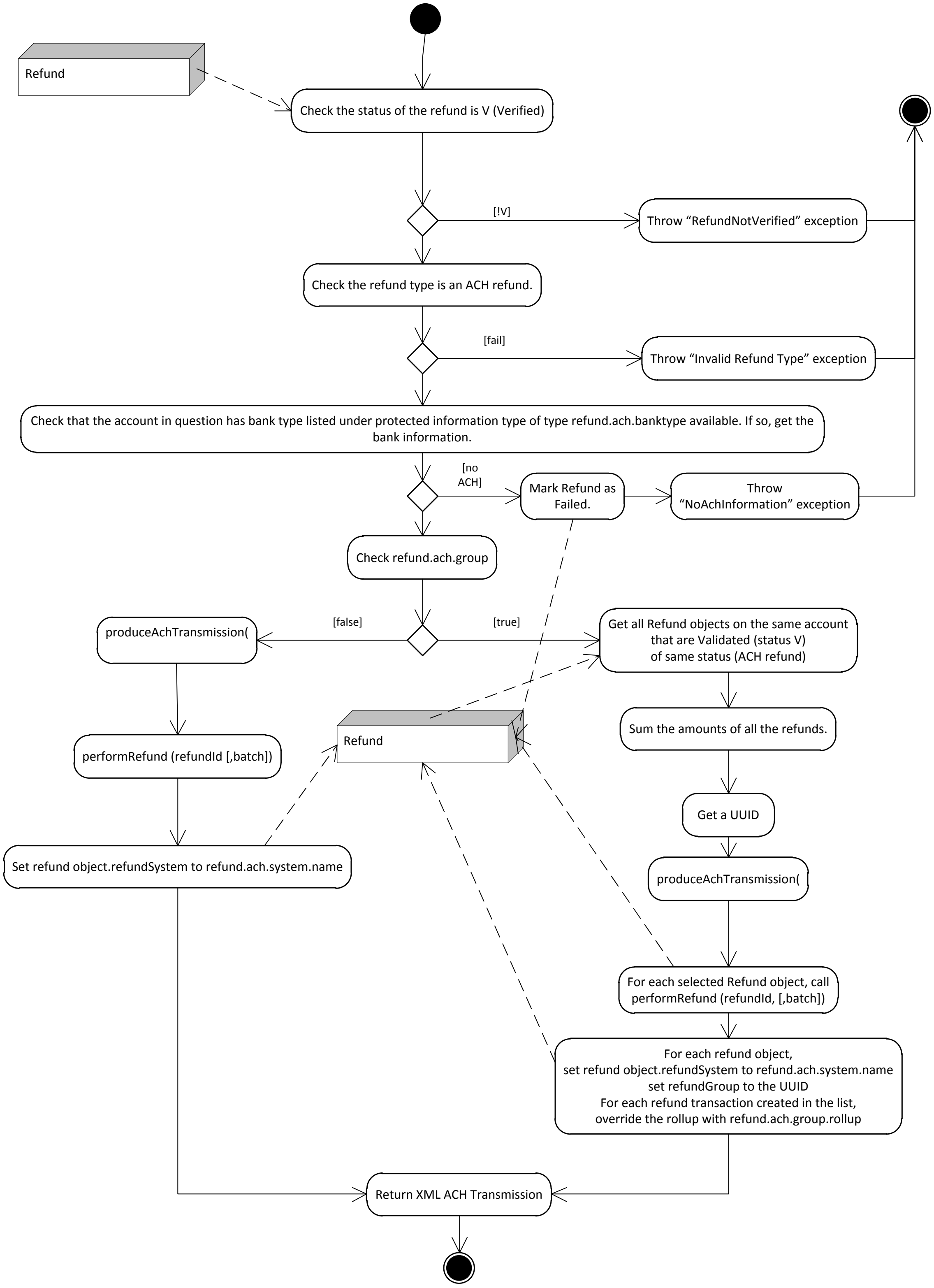

Refund
Check the status of the refund is V (Verified)
[!V]
Throw “RefundNotVerified” exception
Check the refund type is a payoff refund
[fail]
Throw “Invalid Refund Type” exception
perfomRefund(refundId[,batch])
Create a payment on the same account
Of type account.refund.payoff.type in the amount of the refund.
Set refund object.refundSystem to account.refund.system.name
Transaction
Refund
If overrideStatementText is not null, set the statement text for the credit to this text.

doAllPayoffRefunds (batch)

Go through the Refund objects. For each validated refund with type set to payoff refund (refund.payoff.type). For each one that is found, call doPayoffRefund (refundId, batch).

doAchRefund(refundId)

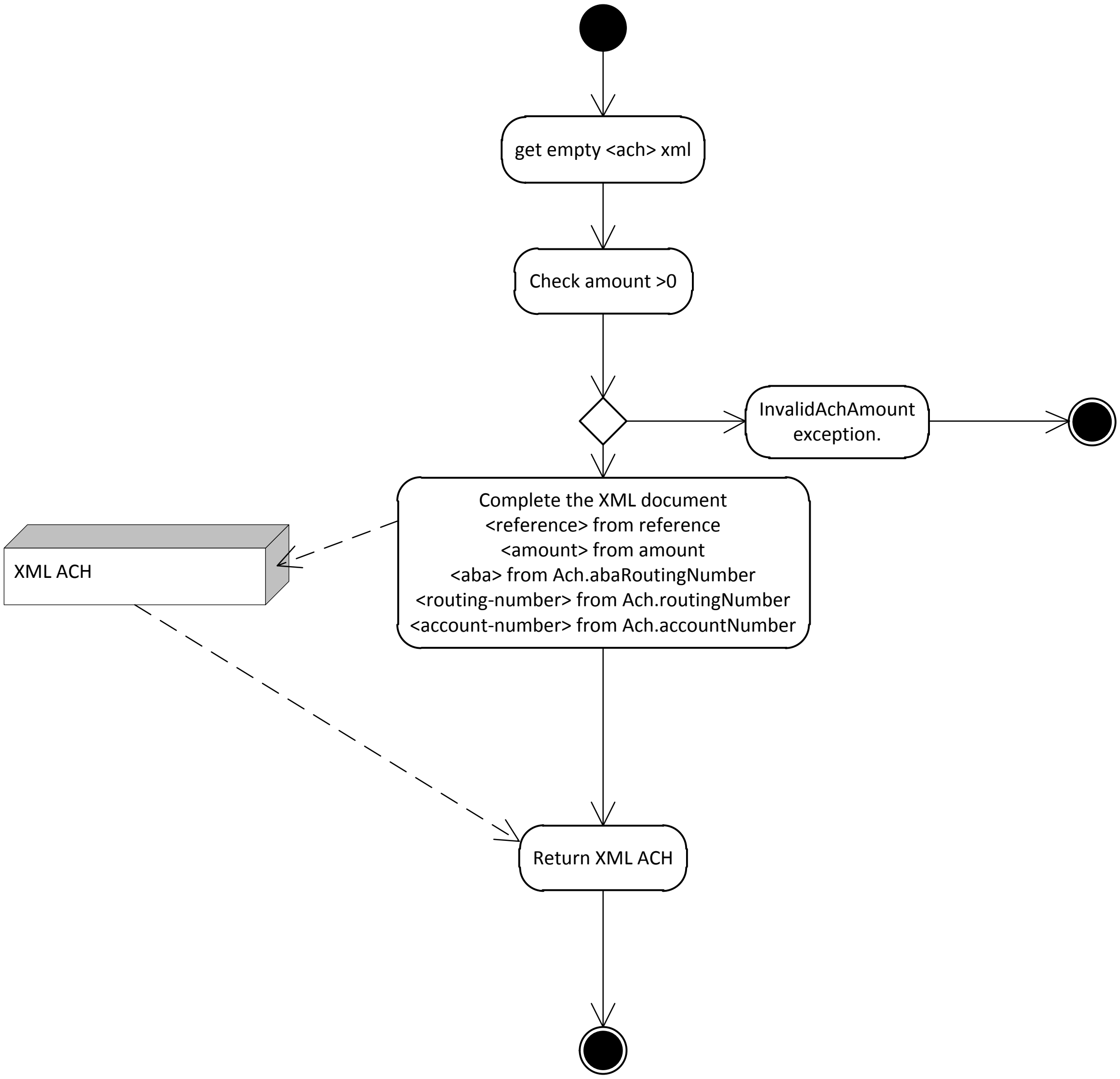
doAchRefund(refundId, batch)


Refund
Check the status of the refund is V (Verified)
[!V]
Throw "RefundNotVerified" exception
Check the refund type is an ACH refund.
[fail]
Throw "Invalid Refund Type" exception
Check that the account in question has bank type listed under protected information type of type refund.ach.banktype available. If so, get the bank information.
[no ACH]
Mark Refund as Failed.
Throw "NoAchInformation" exception
Check refund.ach.group
[false]
[true]
produceAchTransmission(
Get all Refund objects on the same account that are Validated (status V) of same status (ACH refund)
Refund
Sum the amounts of all the refunds.
performRefund (refundId [,batch])
Get a UUID
Set refund object.refundSystem to refund.ach.system.name
produceAchTransmission(
For each selected Refund object, call performRefund (refundId, [,batch])
For each refund object, set refund object.refundSystem to refund.ach.system.name set refundGroup to the UUID For each refund transaction created in the list, override the rollup with refund.ach.group.rollup
Return XML ACH Transmission

doAllAchRefunds (batch)

Iterate through all check refunds (refund.ach.type) that are validated, and produce document of <batch-ach>. Return XML document.

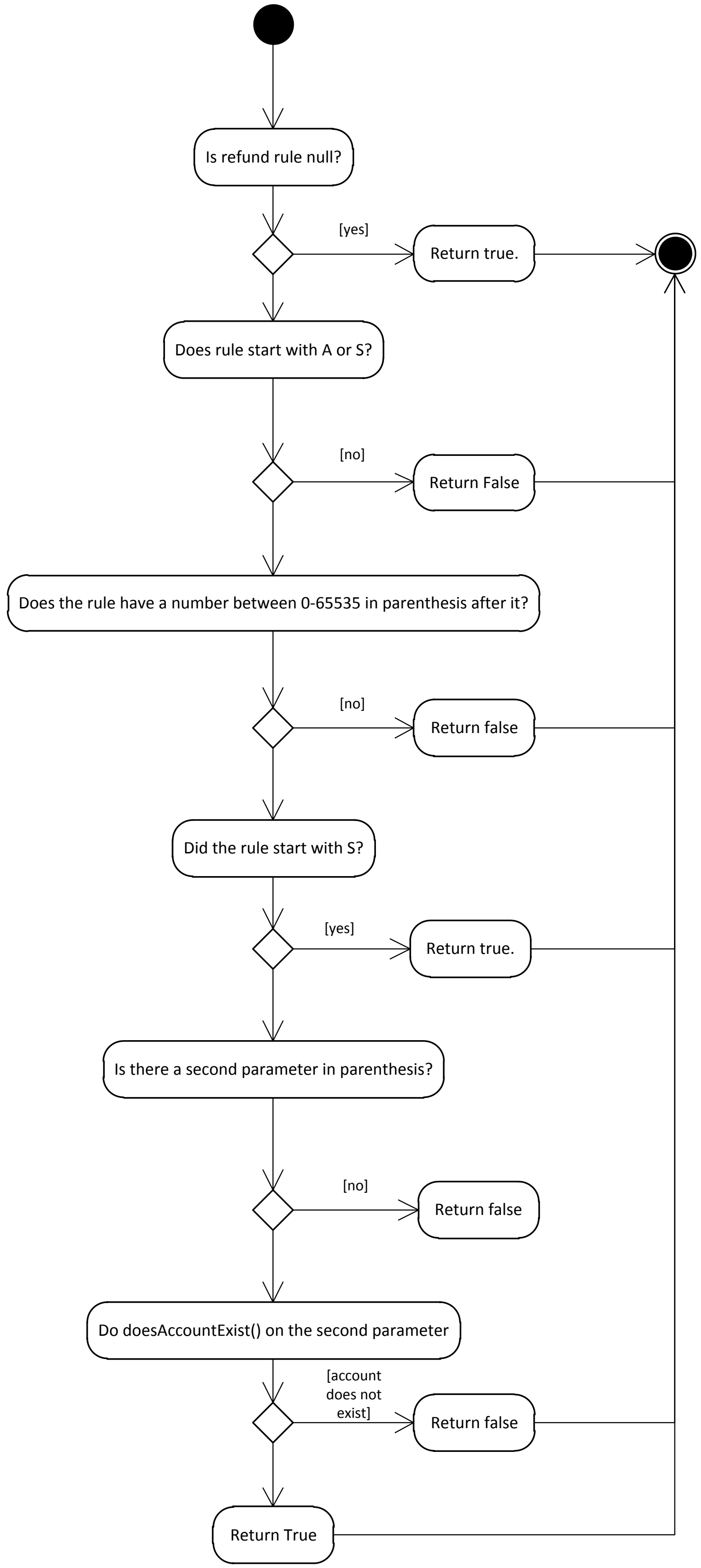
produceAchTransmission (amount, reference, ach)


get empty <ach> xml
Check amount >0
InvalidAchAmount exception.
Complete the XML document
<reference> from reference
<amount> from amount
<aba> from Ach.abaRoutingNumber
<routing-number> from Ach.routingNumber
<account-number> from Ach.accountNumber
XML ACH
Return XML ACH

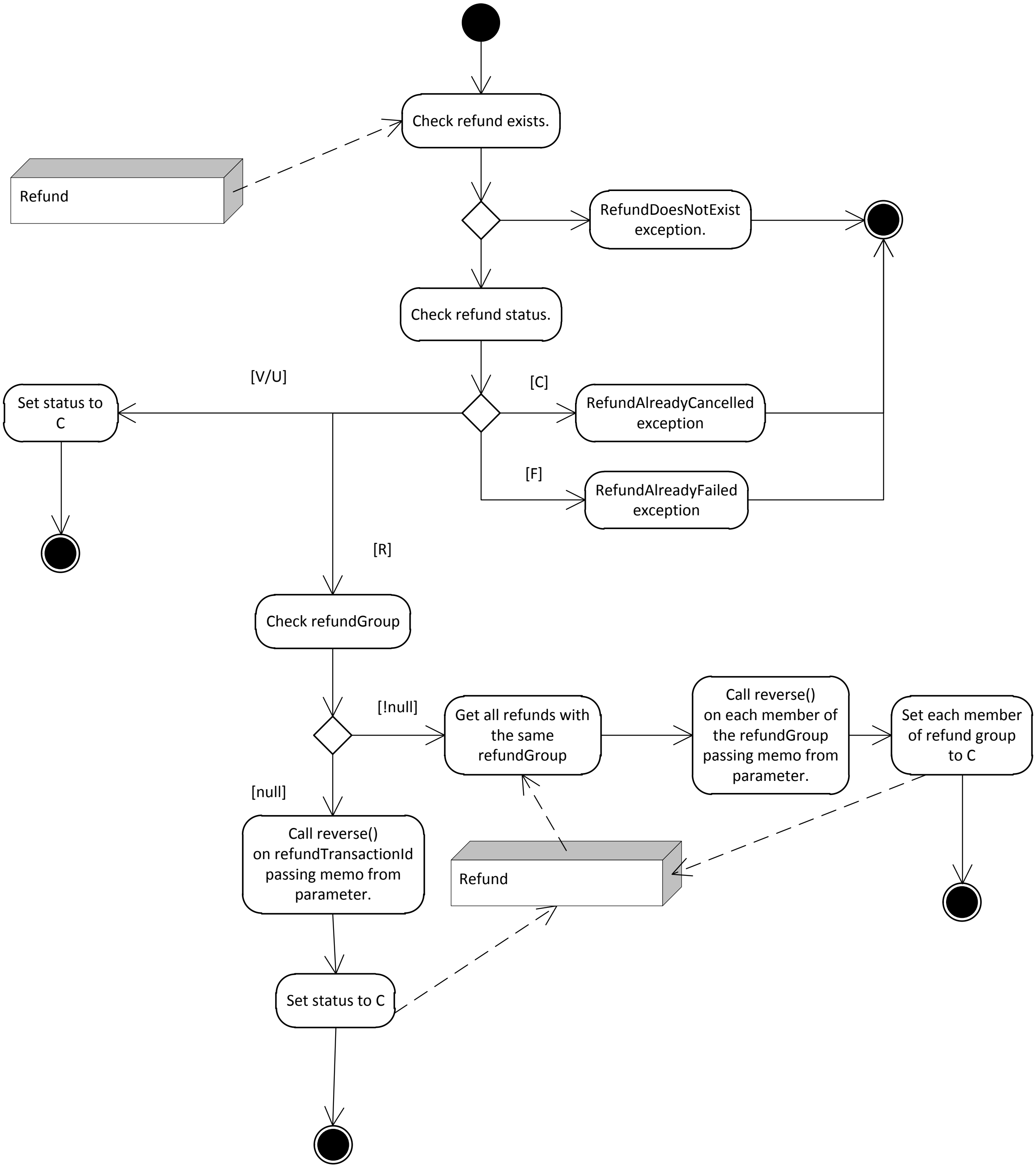
doAllAchRefunds (batch)

Go through the Refund objects. For each validated refund with type set to ach refund (refund.ach.type). For each one that is found, call doAchRefund (refundId, batch).

isRefundRuleValid(refundRule)


Is refund rule null?
[yes]
Return true.
Does rule start with A or S?
[no]
Return False
Does the rule have a number between 0-65535 in parenthesis after it?
[no]
Return false
Did the rule start with S?
[yes]
Return true.
Is there a second parameter in parenthesis?
[no]
Return false
Do doesAccountExist() on the second parameter
[account does not exist]
Return false
Return True

cancelRefund (refundId, memo)

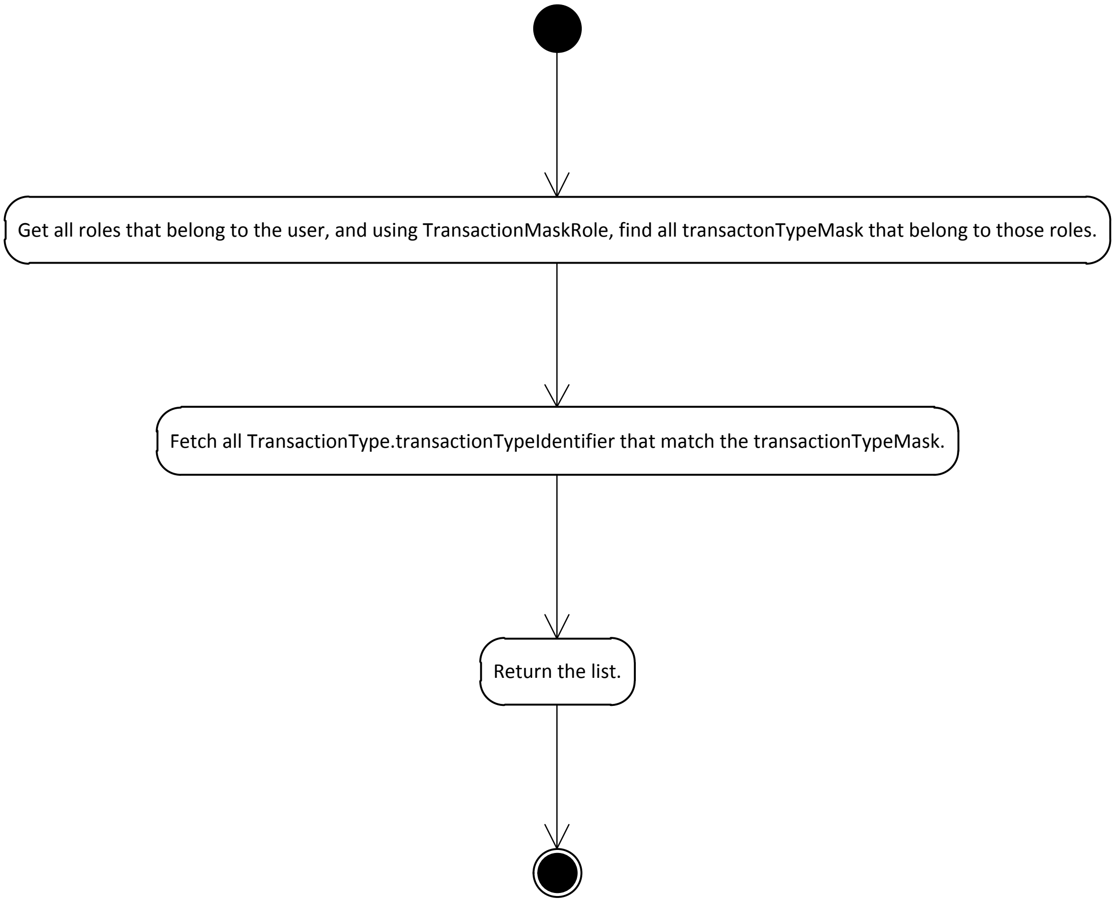

Check refund exists.
Refund
RefundDoesNotExist exception.
Check refund status.
[V/U]
[C]
Set status to C
RefundAlreadyCancelled exception
[F]
RefundAlreadyFailed exception
[R]
Check refundGroup
[!null]
Get all refunds with the same refundGroup
Call reverse() on each member of the refundGroup passing memo from parameter.
Set each member of refund group to C
[null]
Call reverse() on refundTransactionId passing memo from parameter.
Refund
Set status to C

Access Control (Security) Service

The AccessControl service is used to mediate security control between KSA and KIM. All permissions are stored within KIM, allowing easy and standardized access to the KIM permissions system.

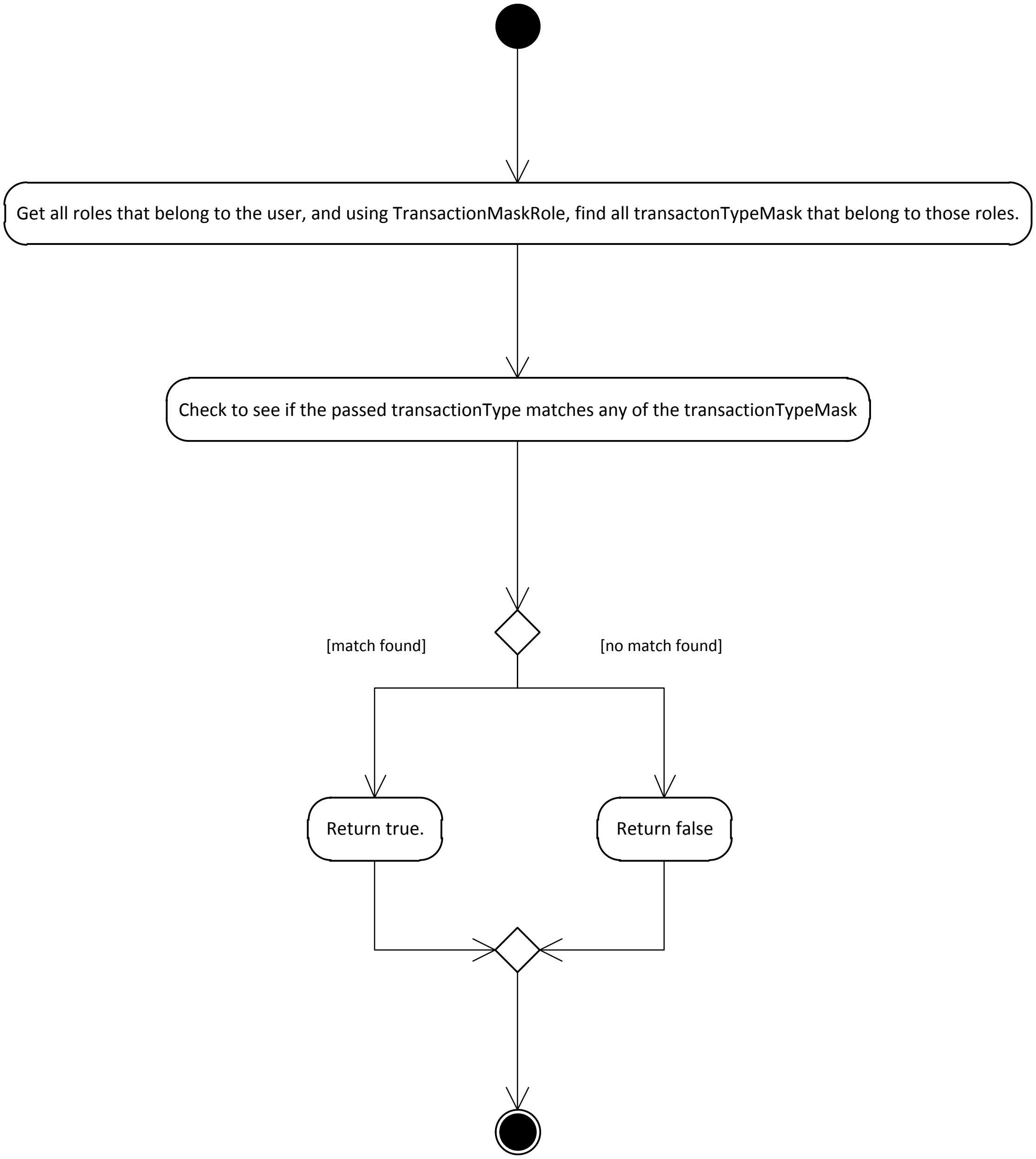
getAllowedTransactionType()

getAllowedTransactionType (entityId)


Get all roles that belong to the user, and using TransactionMaskRole, find all transactonTypeMask that belong to those roles.
Fetch all TransactionType.transactionTypeIdentifier that match the transactionTypeMask.
Return the list.

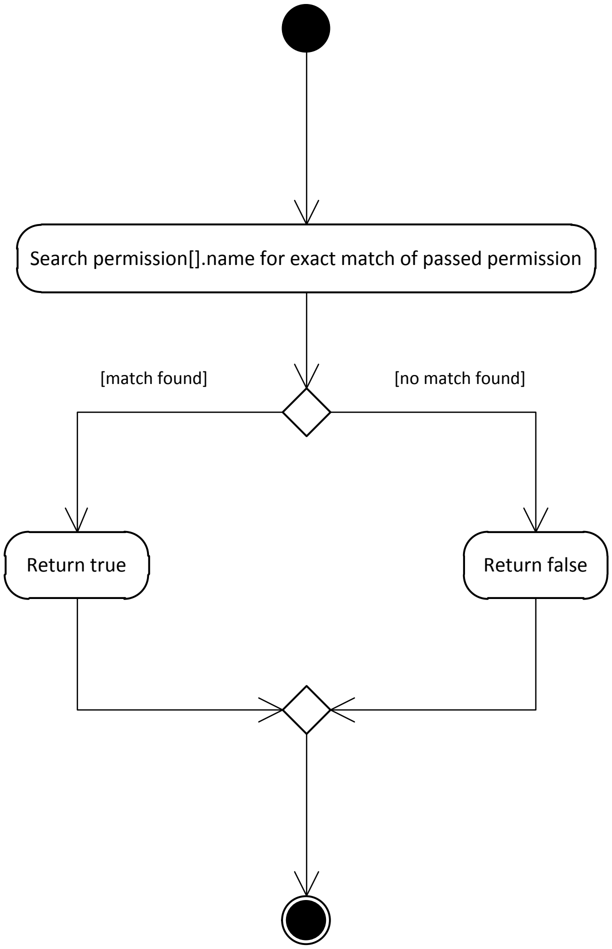
isTransactionTypeAllowed (transactionTypeIdentifier)

isTransactionTypeAllowed (entityId, transactionTypeIdentifier)


Get all roles that belong to the user, and using TransactionMaskRole, find all transactonTypeMask that belong to those roles.
Check to see if the passed transactionType matches any of the transactionTypeMask
[match found]
[no match found]
Return true.
Return false

isAllowed (permissionName)

isAllowed (entityId, permissionName)


Search permission[].name for exact match of passed permission
[match found]
[no match found]
Return true
Return false

refresh()

Forces a reload of the cached TransactionTypePermission class. This should be called after changing a permission if it is required that the change be propagated instantly.

User Preference Service

The user preference service allows the system to store attributes for each user, allowing fine-grained preference control for each user. In many instances, the UI will allow a student to alter preferences stored here. The details of these preferences are located in the document “System-wide Configuration Settings”.

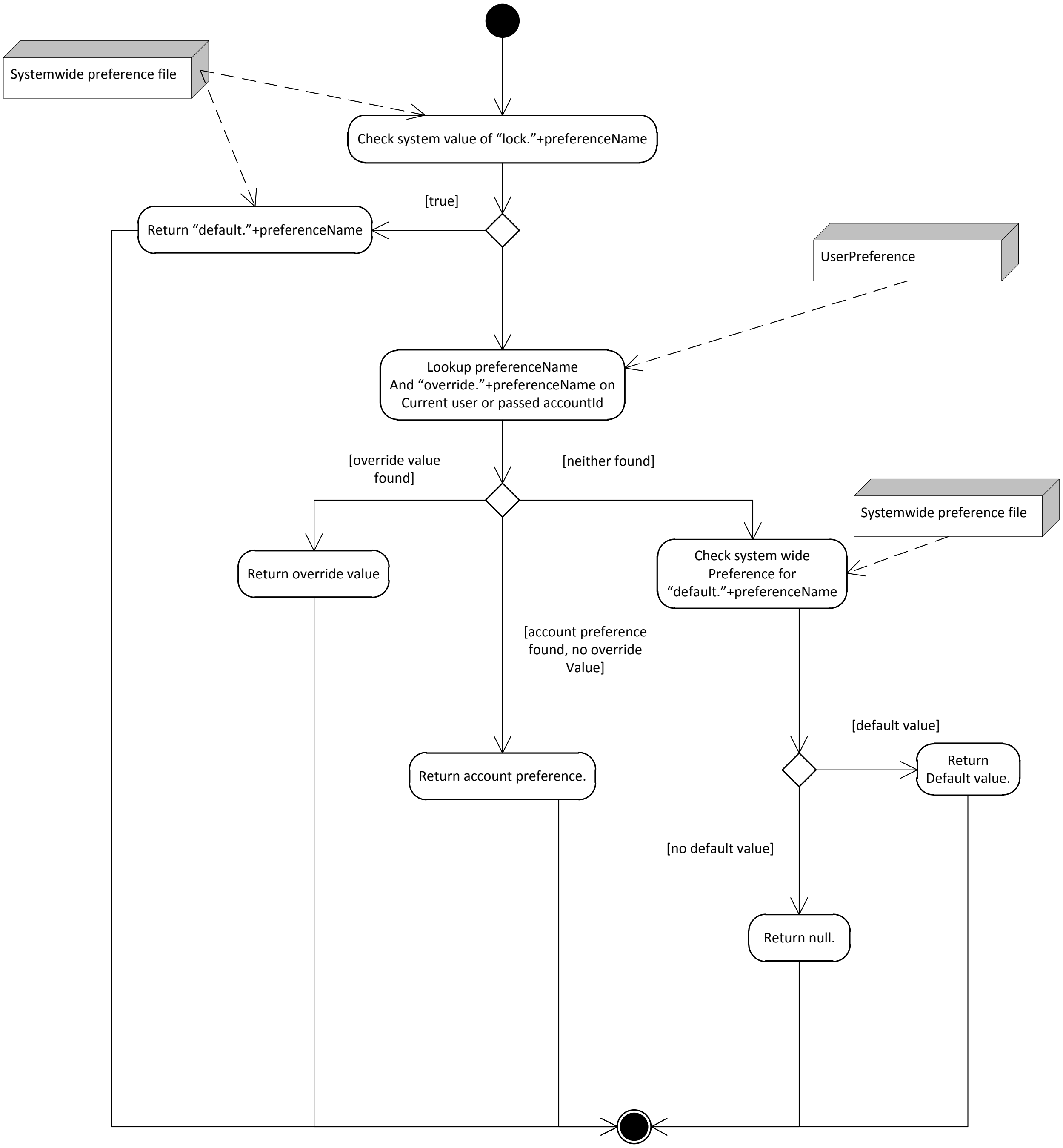
These are simple, but powerful key/pair values. They are called by name. For example, a student may have a preference of how they want to receive their refunds. In this example, this is called refund.method. In certain circumstances, the bursar’s office may wish to override the student’s preference. To do this, they can set the preference override.refund.method, and this will take precedence over refund.method.

If neither of these values are set, then the system will look to the system-wide options for default.refund.method.

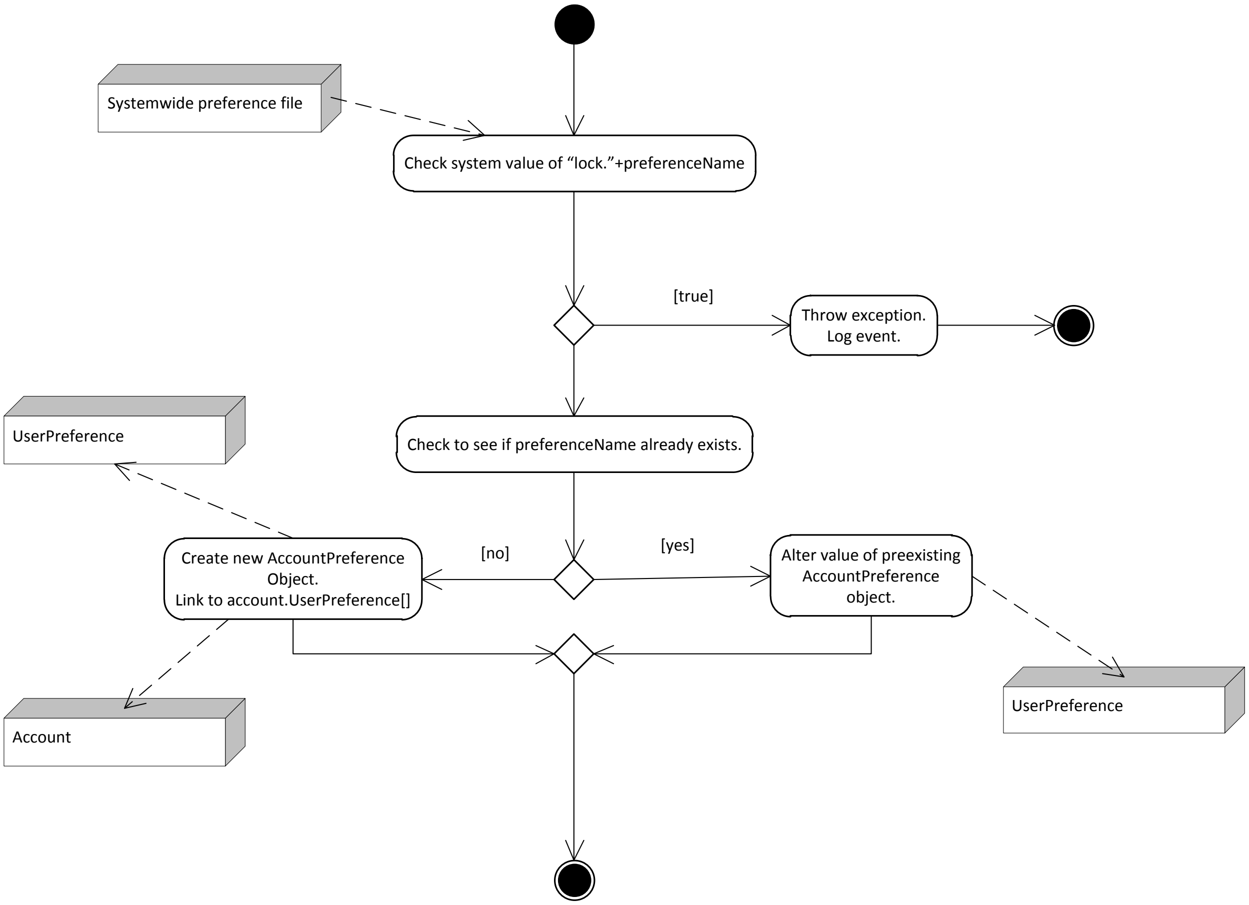
If a school does not want a student to be able to set refund method, then locked.refund.method will be set to true. In this case, default.refund.method will apply no matter what values might be stored in user preferences.

getPreference (preferenceName)

getPreference (preferenceName, accountId)


Systemwide preference file
Check system value of "lock."+preferenceName
[true]
Return "default."+preferenceName
UserPreference
Lookup preferenceName
And "override."+preferenceName on
Current user or passed accountId
[override value
found]
[neither found]
Systemwide preference file
Return override value
Check system wide
Preference for
"default."+preferenceName
[account preference
found, no override
Value]
[default value]
Return account preference.
Return
Default value.
[no default value]
Return null.

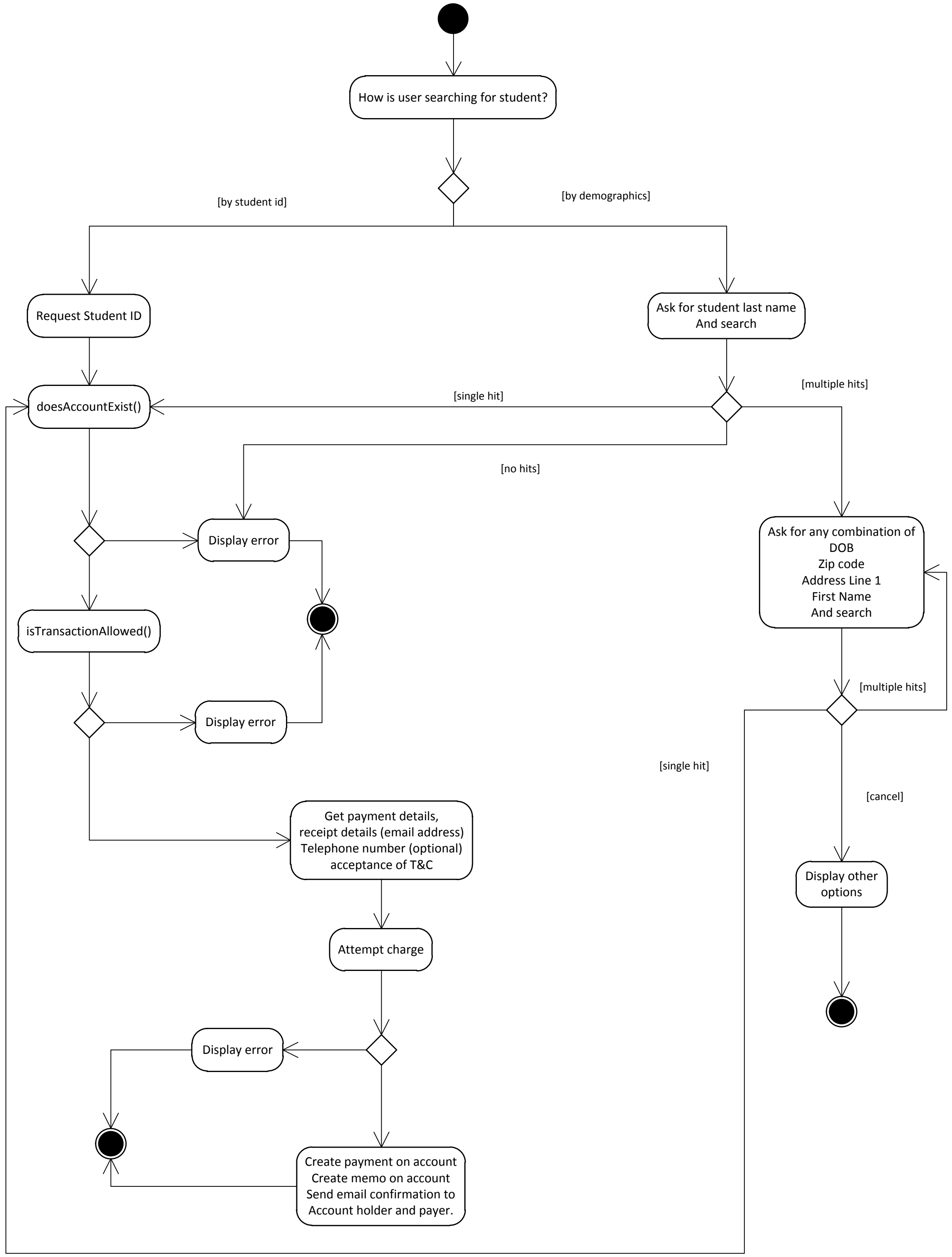
setPreference (preferenceName)
setPreference (preferenceName, accountId)


Systemwide preference file
Check system value of "lock."+preferenceName
[true]
Throw exception.
Log event.
UserPreference
Check to see if preferenceName already exists.
[no]
[yes]
Create new AccountPreference Object.
Link to account.UserPreference[]
Alter value of preexisting AccountPreference object.
Account
UserPreference

Miscellaneous Process Flows

These flows do not try to describe methods, rather larger processes to achieve a result.

Unauthenticated Web Portal Flow

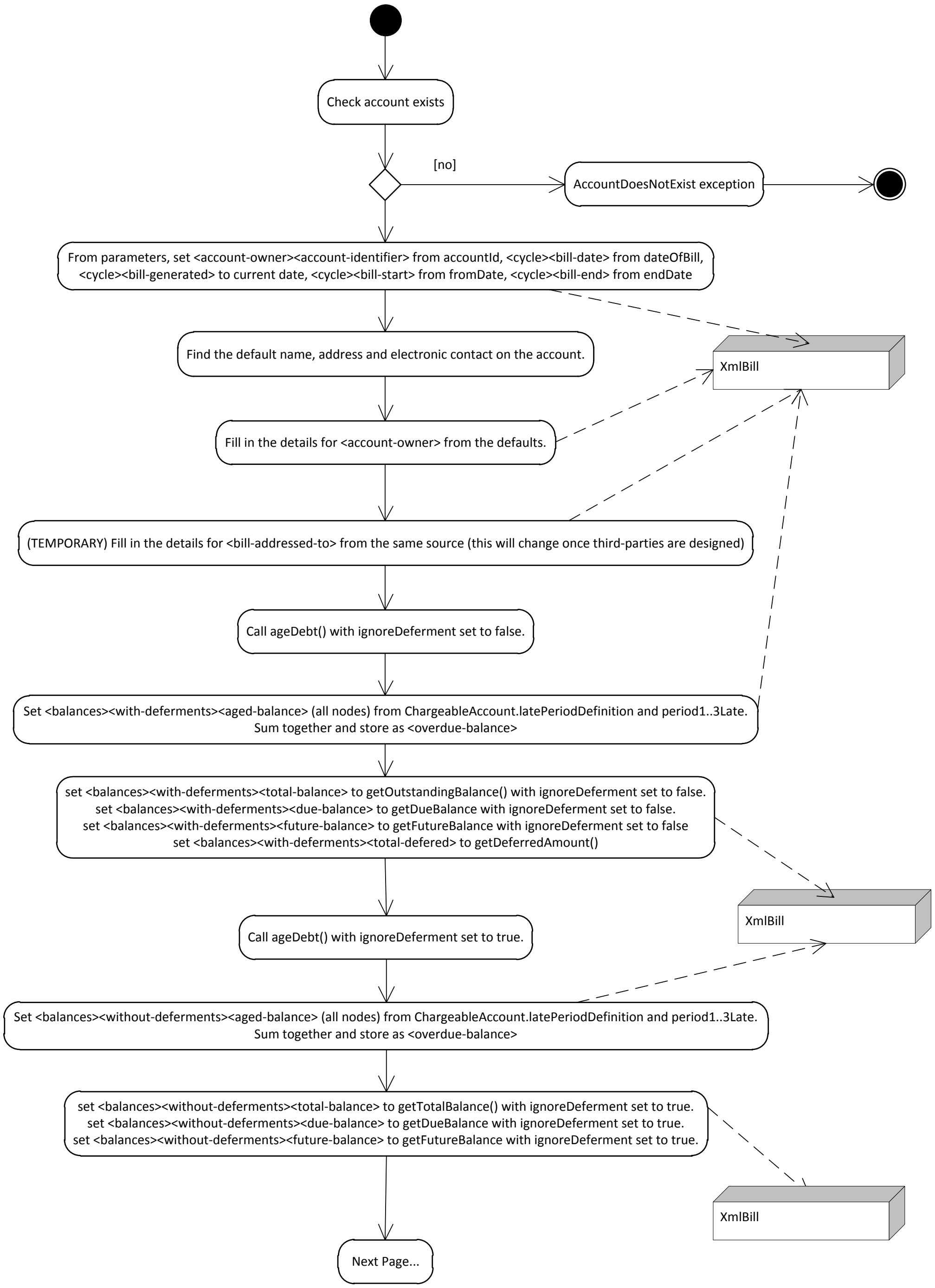

How is user searching for student?
[by student id]
[by demographics]
Request Student ID
Ask for student last name And search
doesAccountExist()
[single hit]
[multiple hits]
[no hits]
Display error
Ask for any combination of DOB Zip code Address Line 1 First Name And search
isTransactionAllowed()
[multiple hits]
Display error
[single hit]
[cancel]
Get payment details, receipt details (email address) Telephone number (optional) acceptance of T&C
Display other options
Attempt charge
Display error
Create payment on account Create memo on account Send email confirmation to Account holder and payer.

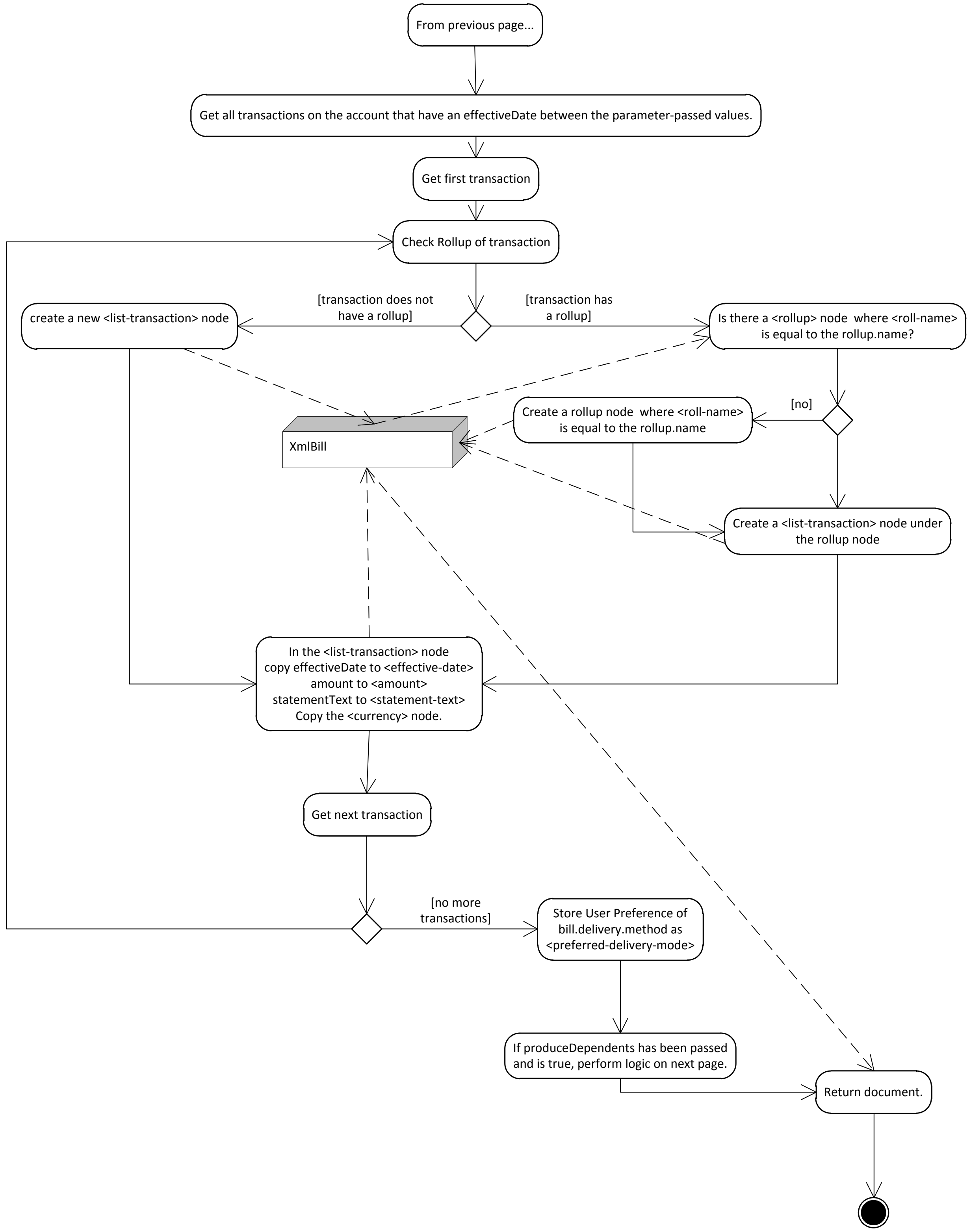
Billing Service

The billing service is responsible for the production of XML bills according the ksa-bill schema. These bills can be exported to other applications to produce bills in whatever format the institution prefers.

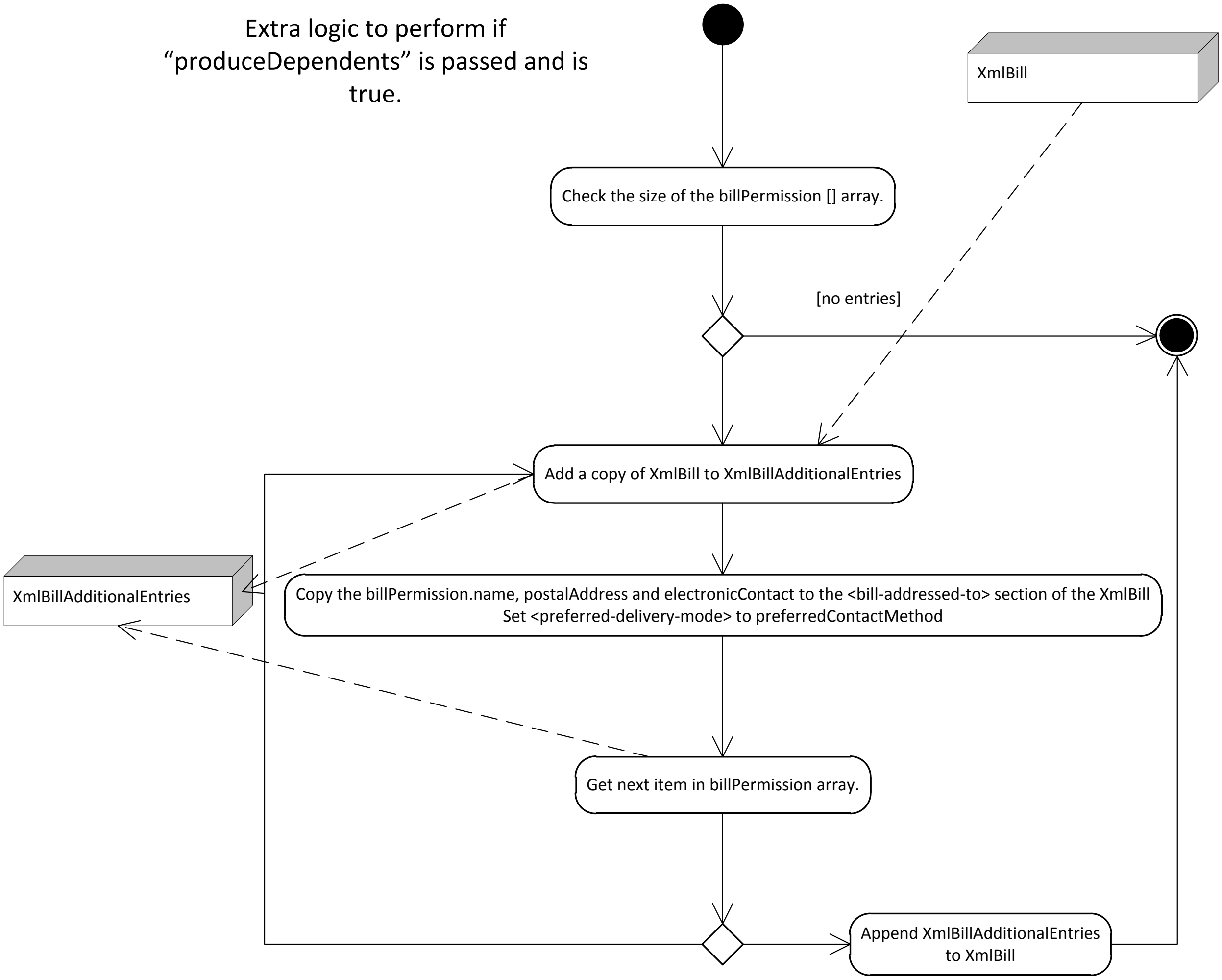
produceBill (accountId, fromDate, toDate, dateOfBill, withDependents)

At this time, only direct-charge account bills are produced by this process flow. In the future, this process will also handle other types of billing.


Check account exists
[no]
AccountDoesNotExist exception
From parameters, set <account-owner><account-identifier> from accountId, <cycle><bill-date> from dateOfBill, <cycle><bill-generated> to current date, <cycle><bill-start> from fromDate, <cycle><bill-end> from endDate
XmlBill
Find the default name, address and electronic contact on the account.
Fill in the details for <account-owner> from the defaults.
(TEMPORARY) Fill in the details for <bill-addressed-to> from the same source (this will change once third-parties are designed)
Call ageDebt() with ignoreDeferment set to false.
Set <balances><with-deferments><aged-balance> (all nodes) from ChargeableAccount.latePeriodDefinition and period1..3Late. Sum together and store as <overdue-balance>
set <balances><with-deferments><total-balance> to getOutstandingBalance() with ignoreDeferment set to false.
set <balances><with-deferments><due-balance> to getDueBalance with ignoreDeferment set to false.
set <balances><with-deferments><future-balance> to getFutureBalance with ignoreDeferment set to false
set <balances><with-deferments><total-defered> to getDeferredAmount()
XmlBill
Call ageDebt() with ignoreDeferment set to true.
Set <balances><without-deferments><aged-balance> (all nodes) from ChargeableAccount.latePeriodDefinition and period1..3Late. Sum together and store as <overdue-balance>
set <balances><without-deferments><total-balance> to getTotalBalance() with ignoreDeferment set to true.
set <balances><without-deferments><due-balance> to getDueBalance with ignoreDeferment set to true.
set <balances><without-deferments><future-balance> to getFutureBalance with ignoreDeferment set to true.
XmlBill
Next Page...


From previous page...
Get all transactions on the account that have an effectiveDate between the parameter-passed values.
Get first transaction
Check Rollup of transaction
[transaction does not have a rollup]
[transaction has a rollup]
create a new <list-transaction> node
Is there a <rollup> node where <roll-name> is equal to the rollup.name?
[no]
Create a rollup node where <roll-name> is equal to the rollup.name
XmlBill
Create a <list-transaction> node under the rollup node
In the <list-transaction> node copy effectiveDate to <effective-date> amount to <amount> statementText to <statement-text> Copy the <currency> node.
Get next transaction
[no more transactions]
Store User Preference of bill.delivery.method as <preferred-delivery-mode>
If produceDependents has been passed and is true, perform logic on next page.
Return document.

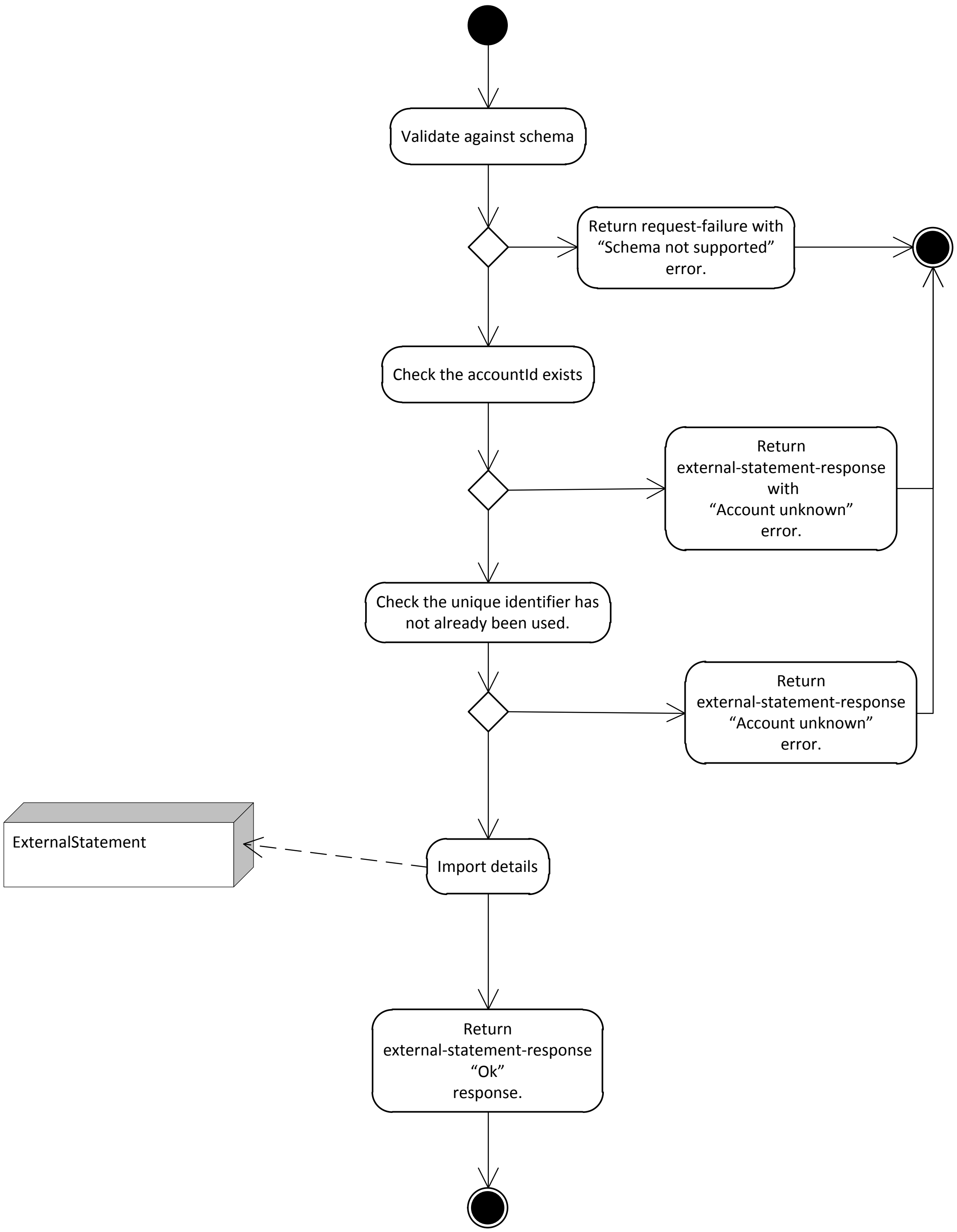
Extra logic to perform if “produceDependents” is passed and is true.


XmlBill
Check the size of the billPermission [] array.
[no entries]
Add a copy of XmlBill to XmlBillAdditionalEntries
XmlBillAdditionalEntries
Copy the billPermission.name, postalAddress and electronicContact to the <bill-addressed-to> section of the XmlBill
Set <preferred-delivery-mode> to preferredContactMethod
Get next item in billPermission array.
Append XmlBillAdditionalEntries to XmlBill

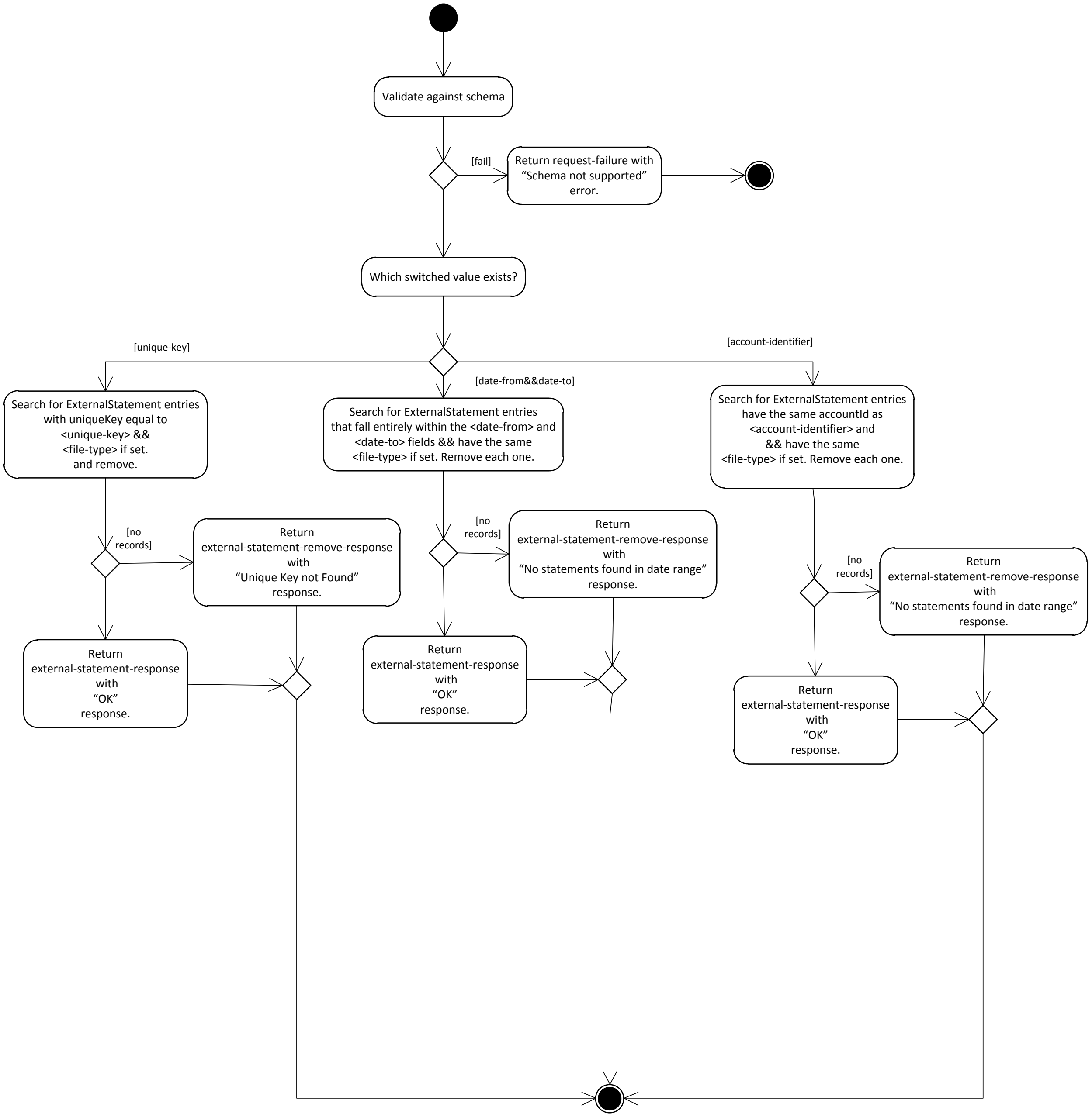
produceBill (list<accountId>, fromDate, toDate, dateOfBill, withDependents)

For each accountId in the list, produce an XML bill and return an XML document containing all the <ksa-bill> nodes.

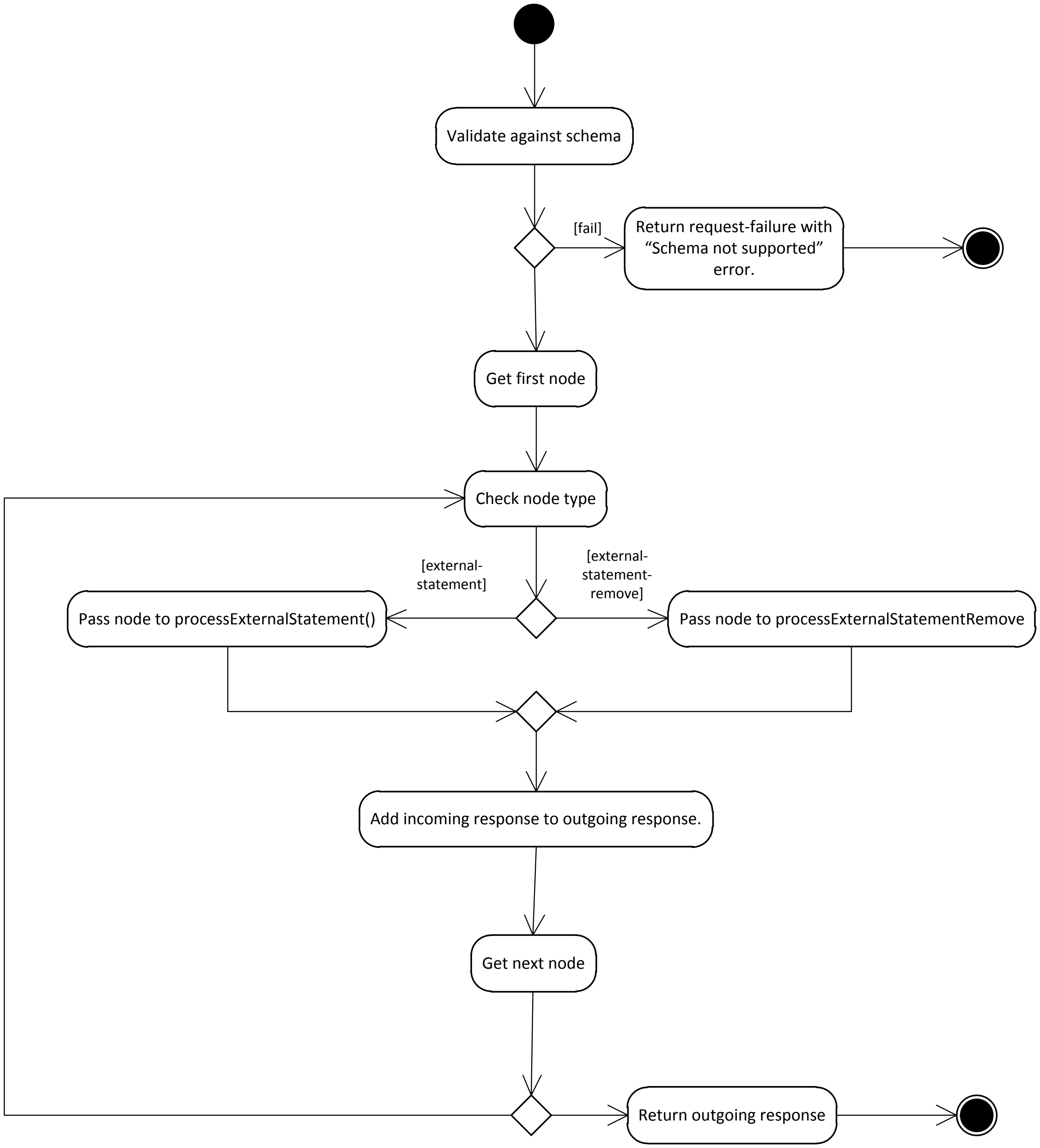
processExternalStatement(xmlMessage)


Validate against schema
Return request-failure with "Schema not supported" error.
Check the accountId exists
Return external-statement-response with "Account unknown" error.
Check the unique identifier has not already been used.
Return external-statement-response "Account unknown" error.
ExternalStatement
Import details
Return external-statement-response "Ok" response.

processExternalStatementRemove (xmlMessage)


Validate against schema
[fail]
Return request-failure with "Schema not supported" error.
Which switched value exists?
[unique-key]
[date-from&&date-to]
[account-identifier]
Search for ExternalStatement entries with uniqueKey equal to <unique-key> && <file-type> if set. and remove.
Search for ExternalStatement entries that fall entirely within the <date-from> and <date-to> fields && have the same <file-type> if set. Remove each one.
Search for ExternalStatement entries have the same accountId as <account-identifier> and && have the same <file-type> if set. Remove each one.
[no records]
Return external-statement-remove-response with "Unique Key not Found" response.
[no records]
Return external-statement-remove-response with "No statements found in date range" response.
[no records]
Return external-statement-remove-response with "No statements found in date range" response.
Return external-statement-response with "OK" response.
Return external-statement-response with "OK" response.
Return external-statement-response with "OK" response.

processExternalStatementBatch (xmlMessage)


Validate against schema
[fail]
Return request-failure with "Schema not supported" error.
Get first node
Check node type
[external-statement]
[external-statement-remove]
Pass node to processExternalStatement()
Pass node to processExternalStatementRemove
Add incoming response to outgoing response.
Get next node
Return outgoing response

Payment Application Service

Payment application services fall under the transaction service, and many payment applications services are already listed under the Transaction and Account services. They are placed here for clarity.

Notice that many of these methods are very explicit and simplified, as they are designed to be called via rules from the rules engine. For this reason, they are rarely overloaded, and the nomenclature of Payments/Charges/Deferments is always used.

paymentApplication (accountId)

Calls the rules set for payment application. Many other services can be used and will be useful to payment application, including a direct creation of an allocation if needed. However, the majority of use cases should be possible by filtering the lists as needed and passing them to the automatic applyPayments() method. This method will create a TransactionList object containing all the unallocated transactions (of any value) for this accountId, and pass this object to the rules engine.

paymentApplication (list<accountId>)

For each accountId, call paymentApplication()

Methods of the TransactionList class (the basis of the Payment Application rules)

Note that many of these methods supply values that can be achieved in other ways, however these methods are designed to be implemented by an end used that has a narrower knowledge of the Java language. Therefore some of the calls are designed to give that user easier access to information that otherwise they would have no way of achieving.

Notice that deferments are always fully allocated by the system, and are therefore not found in the TransactionList supplied to the payment application rules engine.

getNumberOfTransactions ()

Returns the size of the transaction[] array.

refreshList()

Remove all transactions from the list that are now fully allocated (amount = allocatedAmount + lockedAllocationAmount)

getUnallocatedPaymentValue()

getUnallocatedChargeValue()

Returns the value of the unallocated amount of all payments/ charges in the list.

getRestrictedPaymentValue()

getUnrestrictedPaymentValue()

Return the unallocated amount of all restricted or unrestricted payments. Unrestricted payments have a permissableDebitType of "*".

calculateMatrixScore ()

This is a very expensive method, and should be used sparingly.

The matrixTransactionScore is an element that is only calculated during payment application, and changes depending on the list that is passed to this method.

For each credit (payment), this score is the number of debits (charges) IN THE PASSED LIST, which can be paid by this credit. For example, if a payment is unrestricted, then this will equal the number of debits in the list, as it can pay them all. If a payment is restricted to a single debit code, and only one debit with that debit code exists, then its score will be 1.

Conversely, for each debit (charge), the score is the number of credits (payments) that can pay off this debit.

The lower the score, the more limited a credit/debit is. A transaction with a matrixTransactionScore of 0 is not able to be allocated to any transaction in the current working list.

orderByPriority (Boolean ascending)

orderByDate (Boolean ascending)

orderByAmount (Boolean ascending)

orderByUnallocatedAmount (Boolean ascending)

orderByMatrixScore (Boolean ascending)

Order the list on the field mentioned in the method. If ascending is true, sort from lowest to highest, else sort from highest to lowest.

reverseList ()

Invert the list from bottom to top.

getNewList ()

Return a new copy of the list that can be manipulated.

removeCharges()

removePayments()

Remove the appropriate transactions types from the list.

performUnion (transactionList)

Using the list referenced in transaction list, perform a union between the current list and that list. Any entries that are in transactionList that are not in the current list are added to the current list. (OR)

performIntersection (transactionList)

Using the list referenced in transaction list, perform an intersection between the current list and that list. Any entries that are not also in both lists are removed from the current list. (AND)

performCombination (transactionList)

Using the list referenced in transaction list, perform an intersection between the current list and that list. Any items in the passed list will be added to the current list, unless they are already on the list, in which case they will be removed. (XOR).

performSubract (transactionList)

Using the list referenced in transaction list, perform an intersection between the current list and that list. Any items in the passed list will be removed from the current list.

filterByPriority (priorityFrom, priorityTo)

Remove all transactions from the object that fall outside of the priorities specified. If either parameter is null, it can be ignored (no lower/upper value).

filterByDate (dateFrom, dateTo)

Remove all transactions from the object that fall outside of the dates specified. If either parameter is null, it can be ignored (no lower/upper value).

filterByAmount (amountFrom, amountTo)

Remove all transactions from the object that fall outside of the amounts specified. If either parameter is null, it can be ignored (no lower/upper value).

filterByUnallocatedAmount (amountFrom, amountTo)

Remove all transactions from the object that fall outside of the unallocated amounts specified. If either parameter is null, it can be ignored (no lower/upper value).

filterByMatrixScore (matrixScoreFrom, matrixScoreTo)

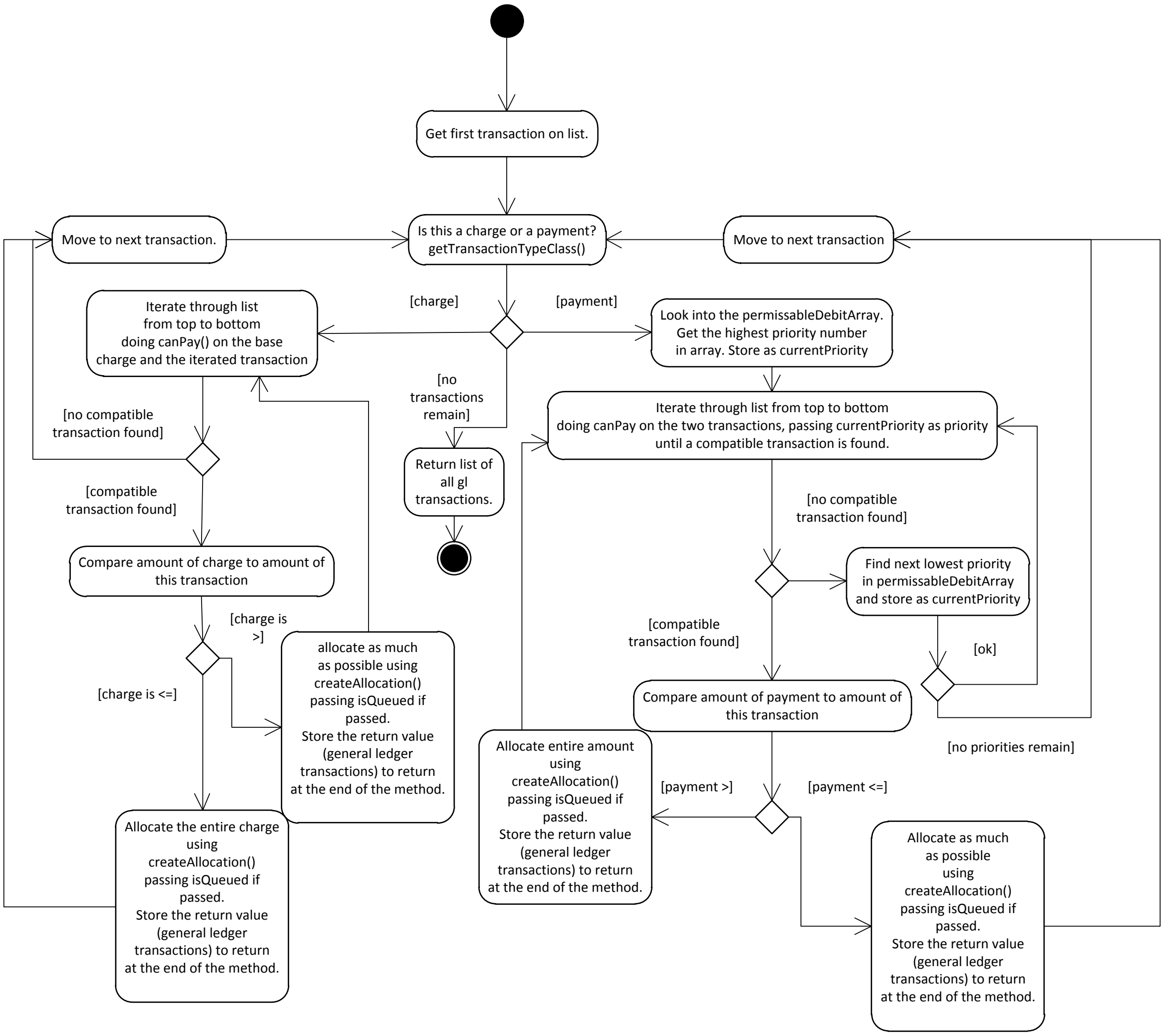
Remove all transactions from the object that fall outside of the unallocated amounts specified. If either parameter is null, it can be ignored (no lower/upper value).

applyPayments ()

applyPayments (isQueued)

applyPayments takes the list that has been manipulated by the other payment application filters. The system will then iterate through the list and apply payments, following simple payment logic. Note that when a payment is encountered, the priority in the permissibleDebitArray will be used. Where a charge is encountered, the next available payment will be applied (that is allowed to pay the charge). Under normal circumstances, payments will be first in the list, unless the user wants to override this behavior.

If isQueued is set as false, the general ledger transactions that are created will be put into status of Waiting, so they will not be transmitted to the general ledger until this status is set to Queued. This will usually be done by passing the list of general ledger transactions to the summarizeGeneralLedgerTransactions() method.


Get first transaction on list.
Move to next transaction.
Is this a charge or a payment? getTransactionTypeClass()
Move to next transaction
[charge]
[payment]
Iterate through list from top to bottom doing canPay() on the base charge and the iterated transaction
Look into the permissableDebitArray. Get the highest priority number in array. Store as currentPriority
[no transactions remain]
Iterate through list from top to bottom doing canPay on the two transactions, passing currentPriority as priority until a compatible transaction is found.
[no compatible transaction found]
Return list of all gl transactions.
[compatible transaction found]
[no compatible transaction found]
Find next lowest priority in permissableDebitArray and store as currentPriority
Compare amount of charge to amount of this transaction
[charge is >]
[compatible transaction found]
[ok]
allocate as much as possible using createAllocation() passing isQueued if passed. Store the return value (general ledger transactions) to return at the end of the method.
[charge is <=]
Compare amount of payment to amount of this transaction
Allocate entire amount using createAllocation() passing isQueued if passed. Store the return value (general ledger transactions) to return at the end of the method.
[no priorities remain]
[payment >]
[payment <=]
Allocate the entire charge using createAllocation() passing isQueued if passed. Store the return value (general ledger transactions) to return at the end of the method.
Allocate as much as possible using createAllocation() passing isQueued if passed. Store the return value (general ledger transactions) to return at the end of the method.

Collections Service (In Progress – This Service is Phase 2: This is designed only to meet phase 1 objective)

assignToCollectionAgency (accountId, collectionAgency, memo)

Calls the rules engine to perform the prerequisite tasks for assigning to collection agency, including establishing flags, blocks, audit trail, and setting the account status to the appropriate value. Finally, this method adds the accountId to the collectionAgency object. Then the memo for this event is placed on the account.

removeFromCollectionAgency (accountId, memo)

Remove the accountId from the collection agency list. This does not knock the account out of "collections" status.

removeFromCollections (accountId, memo)

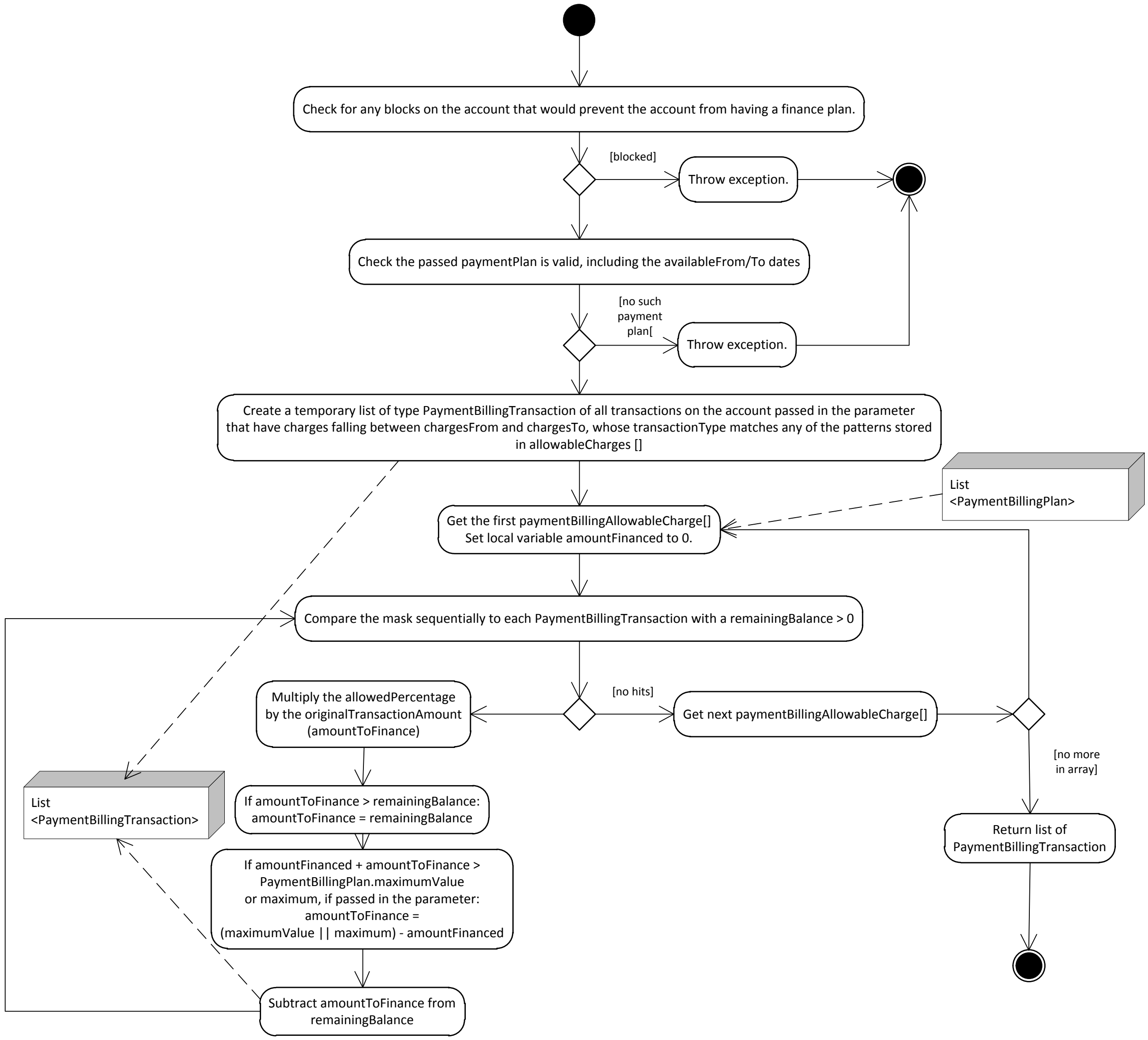
Rule-based process to remove the account from collections status, including lifting of any blocks related to that status, and de-assigning the collection agency if one exists. (see removeFromCollectionAgency()).

Payment Billing Service (Phase 2)

generatePaymentBillingAllowableList (accountId, paymentBillingPlan)

generatePaymentBillingAllowableList (accountId, paymentBillingPlan, maximum)

This method returns a list of type PaymentBillingTransaction, detailing which transactions can be financed, and how much of each of them can be financed. The parameters for this calculation derive from PaymentBillingPlan.


Check for any blocks on the account that would prevent the account from having a finance plan.
[blocked]
Throw exception.
Check the passed paymentPlan is valid, including the availableFrom/To dates
[no such payment plan[
Throw exception.
Create a temporary list of type PaymentBillingTransaction of all transactions on the account passed in the parameter that have charges falling between chargesFrom and chargesTo, whose transactionType matches any of the patterns stored in allowableCharges []
List <PaymentBillingPlan>
Get the first paymentBillingAllowableCharge[] Set local variable amountFinanced to 0.
Compare the mask sequentially to each PaymentBillingTransaction with a remainingBalance > 0
Multiply the allowedPercentage by the originalTransactionAmount (amountToFinance)
[no hits]
Get next paymentBillingAllowableCharge[]
[no more in array]
List <PaymentBillingTransaction>
If amountToFinance > remainingBalance: amountToFinance = remainingBalance
If amountFinanced + amountToFinance > PaymentBillingPlan.maximumValue or maximum, if passed in the parameter: amountToFinance = (maximumValue || maximum) - amountFinanced
Return list of PaymentBillingTransaction
Subtract amountToFinance from remainingBalance

generatePaymentBillingSchedule(paymentBillingTransaction[], paymentBillingPlan)

paymentCalculation (totalAmount, percentage, roundingFactor)

This is used to produce a monthly payment amount, based on the original amount, the percentage and the rounding factor.

Check the rounding factor is only a power of 10 (1, 10,100,1000…) or is zero.

Multiply the totalAmount by the percentage.

If roundingFactor is zero, round the percentage up to the second decimal place.

If the rounding factor is set, round the number up to zero for the number of digits covered.

E.X.

If rounding is 1, the last digit will be zero, so 1029 becomes 1030.

If rounding is 100, the last three digits will be 0, so 1029 becomes 2000.

(Rounding factor is not likely to be anything other than 0,1, or 10)

Return the paymentAmount

System Preference Service

getPreference(preferenceName)

If the preference exists, return the value, otherwise throw an exception.

setPreference (preferenceName, value)

Check the user is permitted to change the system preferences, otherwise throw an exception.

If the preference does not exist, create, and incorporate the creator and creation time.

If the preference exists, update the value, and alter the editor and last update.

Services Service ;)

numberMask (number, digits)

Mask the number passed to only show the last digits as passed in the parameter. The masking character can be found in security.masking.character.

take the value of ssnMask and keep that number of the right-most digits. Replace the other digits with the masking character.

(ex. number = “123456789”, digits=4, security.masking.character=X, return “XXXXX6789”)

Reporting Service (Part of KSA-RR)

prepare1098T (accountId, year, ssnMask)

prepare1098T (accountId, year, ssnMask , noRecord)

prepare1098T (accountId, dateFrom, dateTo, year, ssnMask)

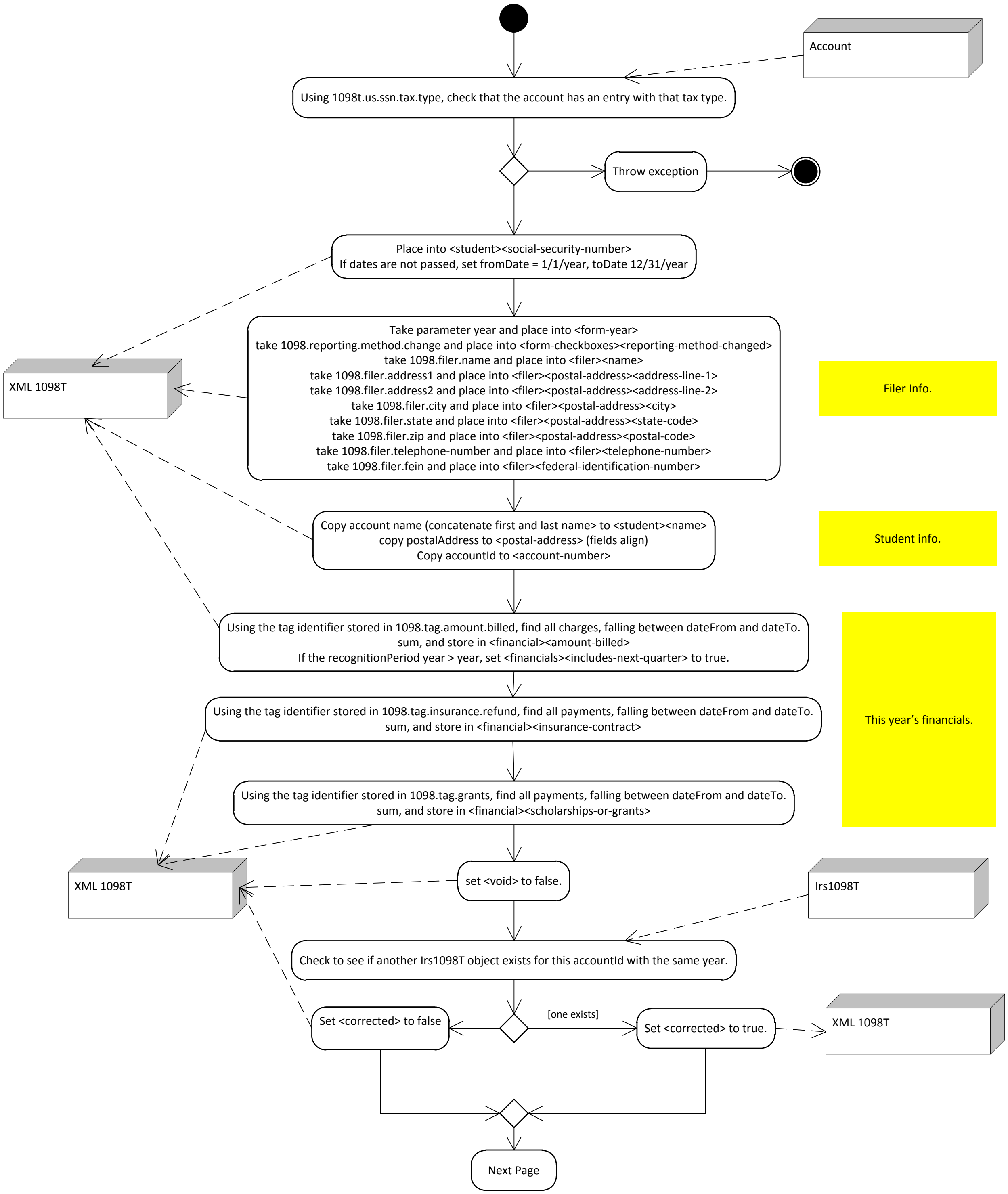
prepare1098T (accountId, dateFrom, dateTo, year, ssnMask , noRecord)

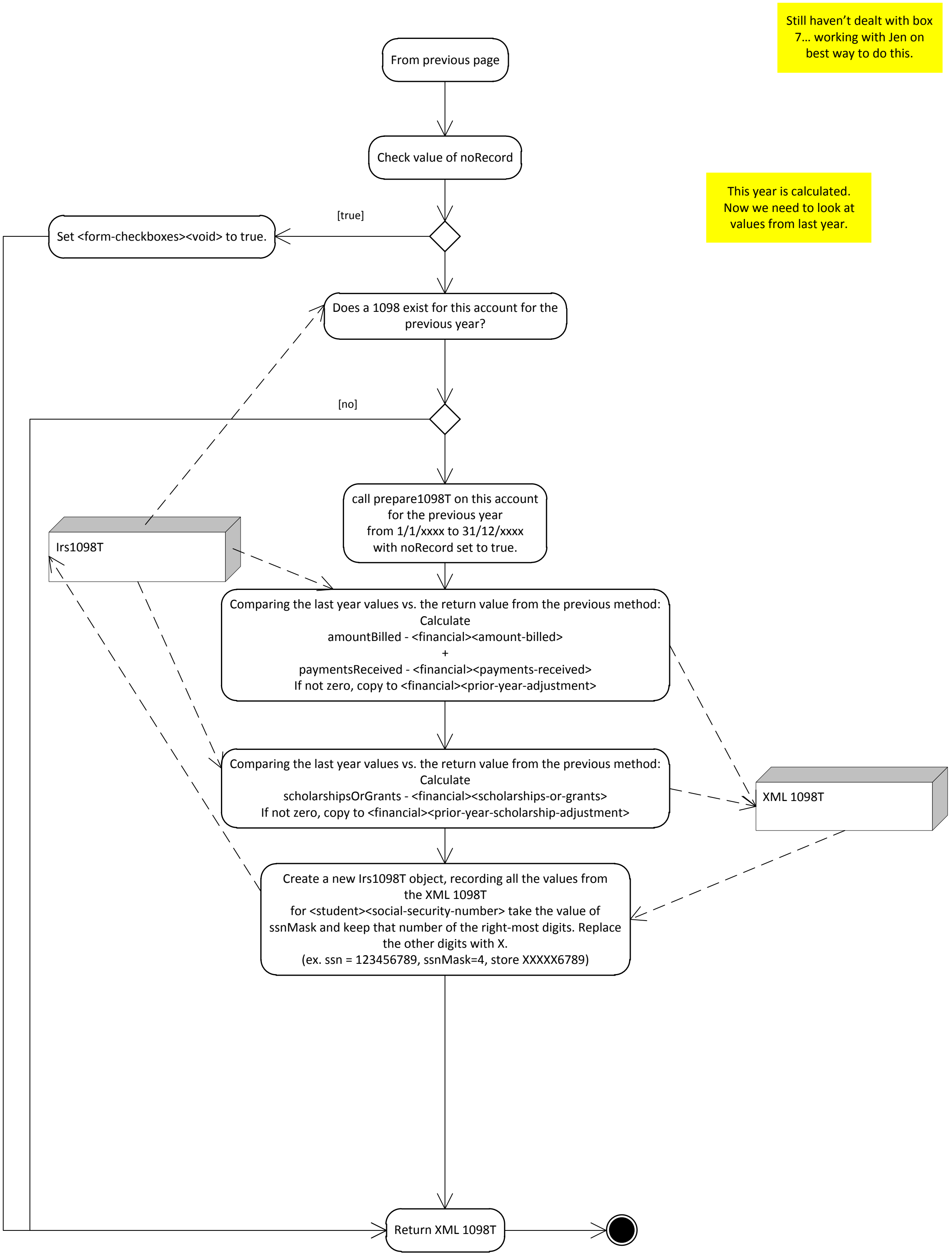
Returns an XML representation of the federal 1098T form. Note that many of the parameters for producing an correct 1098T must be established in the system preferences, and transactions must carry the appropriate tags in order to be counted correctly.

Note that if year is passed, the dateFrom and dateTo fields are assumed to be 1/1/year through 12/31/year. It is unlikely these dates would need to be changed.

ssnMask is the number of final digits of the social security number that will be stored in the local record. The XML file produced will contain the entire social security number, and it is imperative that the institution provide appropriate controls over this data.

If noRecord is passed as true, then the system will not keep a record of the produces 1098T. This option is ONLY to be used if the 1098 is being used to verify previous year’s data, as the system does in order to complete the current year’s return. This form will automatically be returned as void, and a paper 1098T should not be produced from this data.


Account
Using 1098t.us.ssn.tax.type, check that the account has an entry with that tax type.
Throw exception
Place into <student><social-security-number>
If dates are not passed, set fromDate = 1/1/year, toDate 12/31/year
Take parameter year and place into <form-year>
take 1098.reporting.method.change and place into <form-checkboxes><reporting-method-changed>
take 1098.filer.name and place into <filer><name>
take 1098.filer.address1 and place into <filer><postal-address><address-line-1>
take 1098.filer.address2 and place into <filer><postal-address><address-line-2>
take 1098.filer.city and place into <filer><postal-address><city>
take 1098.filer.state and place into <filer><postal-address><state-code>
take 1098.filer.zip and place into <filer><postal-address><postal-code>
take 1098.filer.telephone-number and place into <filer><telephone-number>
take 1098.filer.fein and place into <filer><federal-identification-number>
XML 1098T
Filer Info.
Copy account name (concatenate first and last name> to <student><name>
copy postalAddress to <postal-address> (fields align)
Copy accountId to <account-number>
Student info.
Using the tag identifier stored in 1098.tag.amount.billed, find all charges, falling between dateFrom and dateTo.
sum, and store in <financial><amount-billed>
If the recognitionPeriod year > year, set <financials><includes-next-quarter> to true.
Using the tag identifier stored in 1098.tag.insurance.refund, find all payments, falling between dateFrom and dateTo.
sum, and store in <financial><insurance-contract>
This year's financials.
Using the tag identifier stored in 1098.tag.grants, find all payments, falling between dateFrom and dateTo.
sum, and store in <financial><scholarships-or-grants>
XML 1098T
set <void> to false.
Irs1098T
Check to see if another Irs1098T object exists for this accountId with the same year.
Set <corrected> to false
[one exists]
Set <corrected> to true.
XML 1098T
Next Page

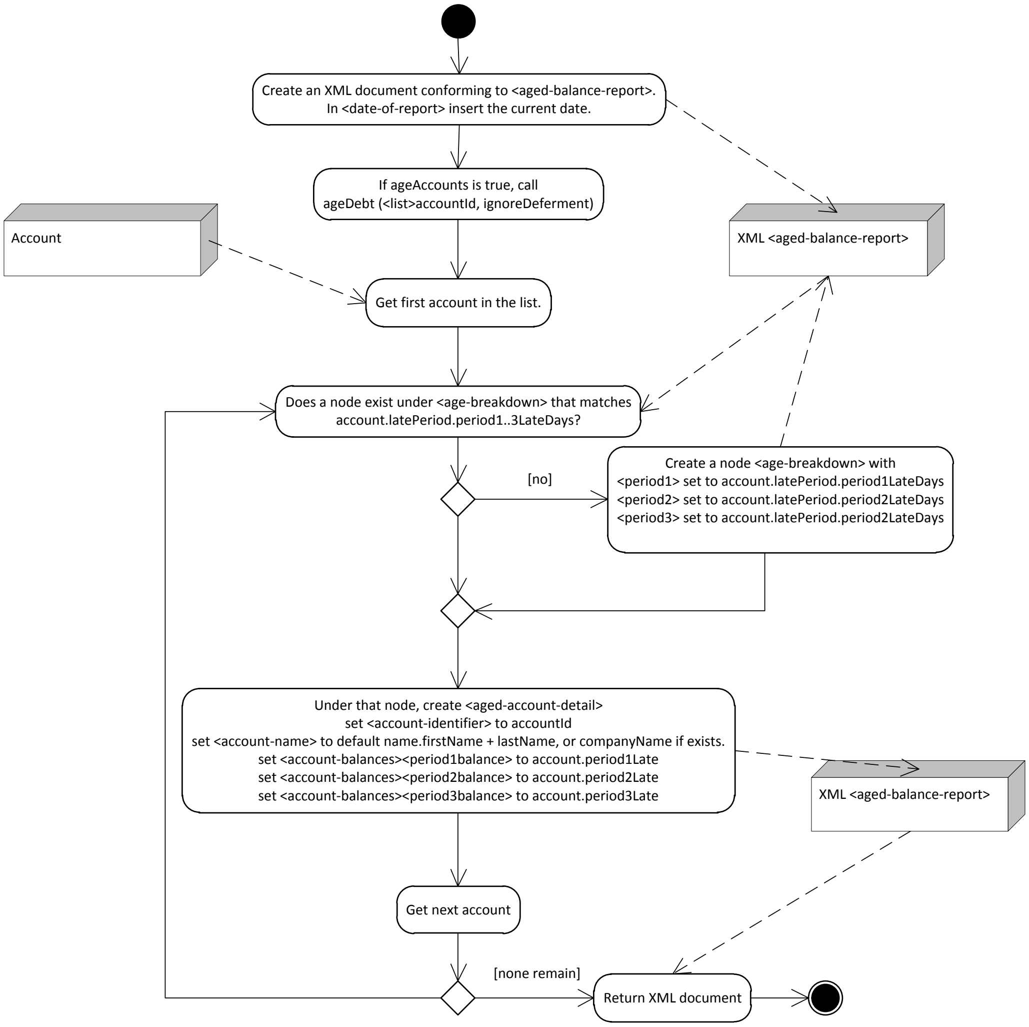

From previous page
Check value of noRecord
[true]
Set <form-checkboxes><void> to true.
Does a 1098 exist for this account for the previous year?
[no]
call prepare1098T on this account for the previous year from 1/1/xxxx to 31/12/xxxx with noRecord set to true.
Irs1098T
Comparing the last year values vs. the return value from the previous method: Calculate amountBilled - <financial><amount-billed> + paymentsReceived - <financial><payments-received> If not zero, copy to <financial><prior-year-adjustment>
Comparing the last year values vs. the return value from the previous method: Calculate scholarshipsOrGrants - <financial><scholarships-or-grants> If not zero, copy to <financial><prior-year-scholarship-adjustment>
XML 1098T
Create a new Irs1098T object, recording all the values from the XML 1098T for <student><social-security-number> take the value of ssnMask and keep that number of the right-most digits. Replace the other digits with X. (ex. ssn = 123456789, ssnMask=4, store XXXXX6789)
Return XML 1098T

Still haven't dealt with box 7… working with Jen on best way to do this.

This year is calculated. Now we need to look at values from last year.

produceAgedBalanceReport (list<accountId>, ignoreDeferments, ageAccounts)

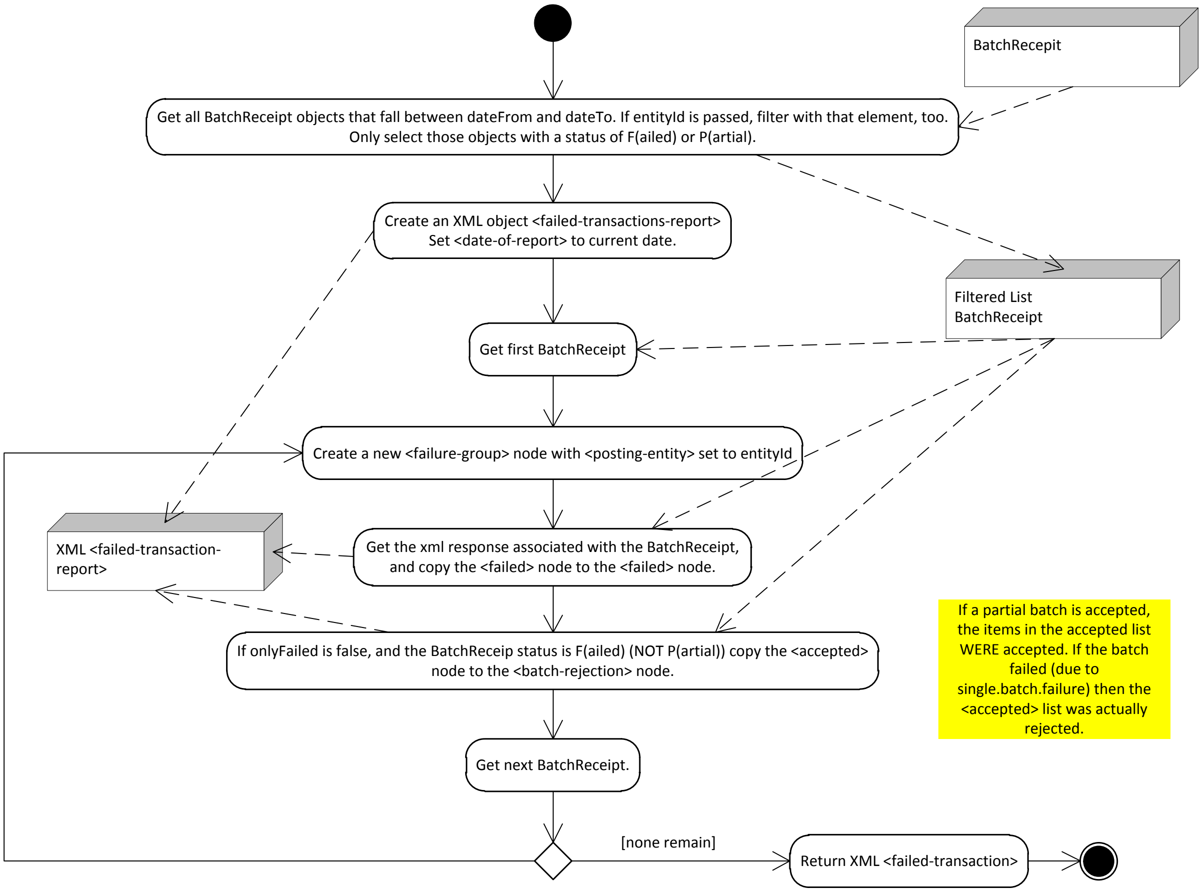
Produce an XML aged balance report for the accounts in the list. If ageAccounts is true, then each account will be aged before the report is run. If ignoreDeferments is true, then deferments will be ignored in the calculation of the balances, ONLY if ageAccounts is set to true.


Create an XML document conforming to <aged-balance-report>.
In <date-of-report> insert the current date.
If ageAccounts is true, call
ageDebt (<list>accountId, ignoreDeferment)
Account
XML <aged-balance-report>
Get first account in the list.
Does a node exist under <age-breakdown> that matches
account.latePeriod.period1..3LateDays?
[no]
Create a node <age-breakdown> with
<period1> set to account.latePeriod.period1LateDays
<period2> set to account.latePeriod.period2LateDays
<period3> set to account.latePeriod.period2LateDays
Under that node, create <aged-account-detail>
set <account-identifier> to accountId
set <account-name> to default name.firstName + lastName, or companyName if exists.
set <account-balances><period1balance> to account.period1Late
set <account-balances><period2balance> to account.period2Late
set <account-balances><period3balance> to account.period3Late
XML <aged-balance-report>
Get next account
[none remain]
Return XML document

produceFailedTransactionReport (dateFrom, dateTo, onlyFailed)

produceFailedTransactionReport (dateFrom, dateTo, entityId, onlyFailed)

Checks the batch records for transactions that have been sent and rejected. The method can filter by date range, and by entity. If no entity is passed, all batches in the date range will be included. If onlyFailed is set to true, only the transactions that failed on their own will be reported (i.e. if a whole batch was rejected due to a failure, the transactions that would have posted ok will not be reported.) If onlyFailed is true, all transactions that were not accepted, even those which were inherently "valid" will be reported.

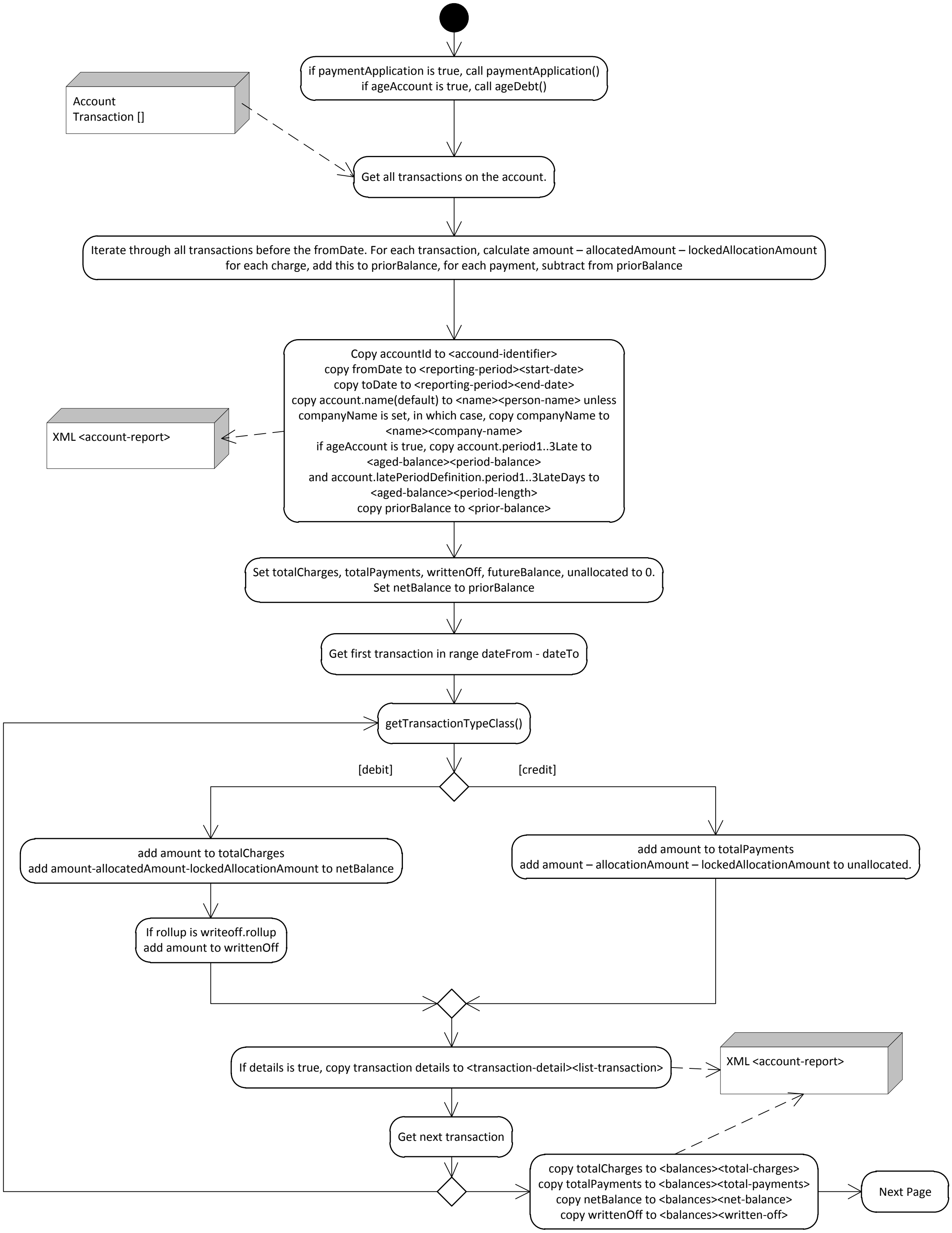

BatchRecepit
Get all BatchReceipt objects that fall between dateFrom and dateTo. If entityId is passed, filter with that element, too. Only select those objects with a status of F(ailed) or P(artial).
Create an XML object <failed-transactions-report> Set <date-of-report> to current date.
Filtered List BatchReceipt
Get first BatchReceipt
Create a new <failure-group> node with <posting-entity> set to entityId
XML <failed-transaction-report>
Get the xml response associated with the BatchReceipt, and copy the <failed> node to the <failed> node.
If onlyFailed is false, and the BatchReceip status is F(ailed) (NOT P(artial)) copy the <accepted> node to the <batch-rejection> node.
If a partial batch is accepted, the items in the accepted list WERE accepted. If the batch failed (due to single.batch.failure) then the <accepted> list was actually rejected.
Get next BatchReceipt.
[none remain]
Return XML <failed-transaction>

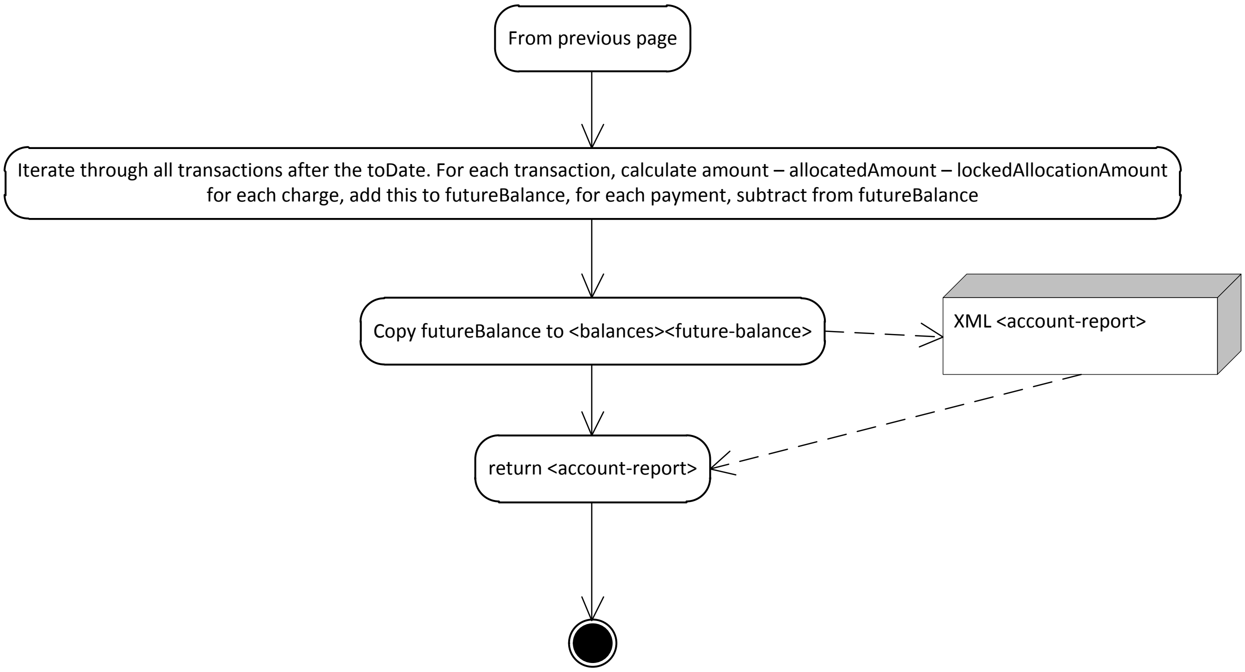
produceAccountReport (list<accountId), fromDate, toDate, details, paymentApplication, ageAccount)

Call produceAccountReport for each account in the list. Return a copy of all the reports.

produceAccountReport (accountId, fromDate, toDate, details, paymentApplication, ageAccount)

Return an XML account report for the account id, between the dates listed. If details is true, then all transactions will also be reported. If paymentApplication is true then paymentApplication() will be run before the account is reported. If ageAccount is true, the account will be aged before the report is generated.


if paymentApplication is true, call paymentApplication()
if ageAccount is true, call ageDebt()
Account
Transaction []
Get all transactions on the account.
Iterate through all transactions before the fromDate. For each transaction, calculate amount – allocatedAmount – lockedAllocationAmount
for each charge, add this to priorBalance, for each payment, subtract from priorBalance
Copy accountId to <accound-identifier>
copy fromDate to <reporting-period><start-date>
copy toDate to <reporting-period><end-date>
copy account.name(default) to <name><person-name> unless
companyName is set, in which case, copy companyName to
<name><company-name>
if ageAccount is true, copy account.period1..3Late to
<aged-balance><period-balance>
and account.latePeriodDefinition.period1..3LateDays to
<aged-balance><period-length>
copy priorBalance to <prior-balance>
XML <account-report>
Set totalCharges, totalPayments, writtenOff, futureBalance, unallocated to 0.
Set netBalance to priorBalance
Get first transaction in range dateFrom - dateTo
getTransactionTypeClass()
[debit]
[credit]
add amount to totalCharges
add amount-allocatedAmount-lockedAllocationAmount to netBalance
add amount to totalPayments
add amount – allocationAmount – lockedAllocationAmount to unallocated.
If rollup is writeoff.rollup
add amount to writtenOff
If details is true, copy transaction details to <transaction-detail><list-transaction>
XML <account-report>
Get next transaction
copy totalCharges to <balances><total-charges>
copy totalPayments to <balances><total-payments>
copy netBalance to <balances><net-balance>
copy writtenOff to <balances><written-off>
Next Page


From previous page
Iterate through all transactions after the toDate. For each transaction, calculate amount – allocatedAmount – lockedAllocationAmount for each charge, add this to futureBalance, for each payment, subtract from futureBalance
Copy futureBalance to <balances><future-balance>
XML <account-report>
return <account-report>