Sigma Systems, Inc.

Process Diagrams & Descriptions for KSA

Sigma Systems
March 2012

Process Models: KSA

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 persistTransaction() to call checkCashLimit() method 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()
Paul	8/28/2012	Corrections to createAllocation(), createLockedAllocation(), removeAllocation(), removeLockedAllocation() to remove problematic reference to GeneralLedgerBreakDown.
Paul	8/30/2012	Minor change to canPay() to take deferments into account.
Paul	8/30/2012	Added allocateReveral() to the payment application methods.
Paul	9/4/2012	Started importStudentProfile() under fee management.
Paul	9/6/2012	Added glOperation to addGeneralLedgerTransaction(). Changes to general ledger processes to take account of glOperation and closed a potential hole in the reconigitionPeriod system under generateGeneralLedgerTransmission()
Paul	9/7/2012	Created the general ledger export processes.
Paul	9/14/2012	Tweak to GL methods to remove recognitionPeriod from the individual transactions.
Paul	9/21/2012	Added all the methods of the transaction type service.
Paul	9/24/2012	Completed the transaction type service, added the tag service. Additional methods to the transaction service to permit addition and removal of rollups and tags at the service layer. Changed remaining “does***()” methods to ***+verb to standardize names.
Paul	9/25/2012	Working on Transfer Transaction services, and reversals.
Paul	9/26/2012	AuditiableEntity methods. Minor changes to removeKeyPair methods per Sergey.
Paul	9/26/2012	Major update to deferment logic.
Paul	9/27/2012	Alterations to FeeBase methods, per Sergey.

Paul	9/27/2012	Added contestCharge() to mimic original deferment logic.
Paul	10/1/2012	Document rework, validating against code created.
Paul	10/3/2012	Completed document rework.
Paul	10/4/2012	Tweak to canPay()
Paul	10/10/2012	Tweak to createTransaction() and persistTransaction()
Paul	10/22/2012	Added cancellationRule methods. Tweak to createTransaction() to calculate the cancellationRule attribute. Added persistTransactionType() method.
Paul	10/23/2012	GeneralLedgerType methods, plus generalLedgerType validation methods. cancelTransaction()
Paul	11/7/2012	Minor change to doAccountRefund() to store the outgoing transaction during an account refund.

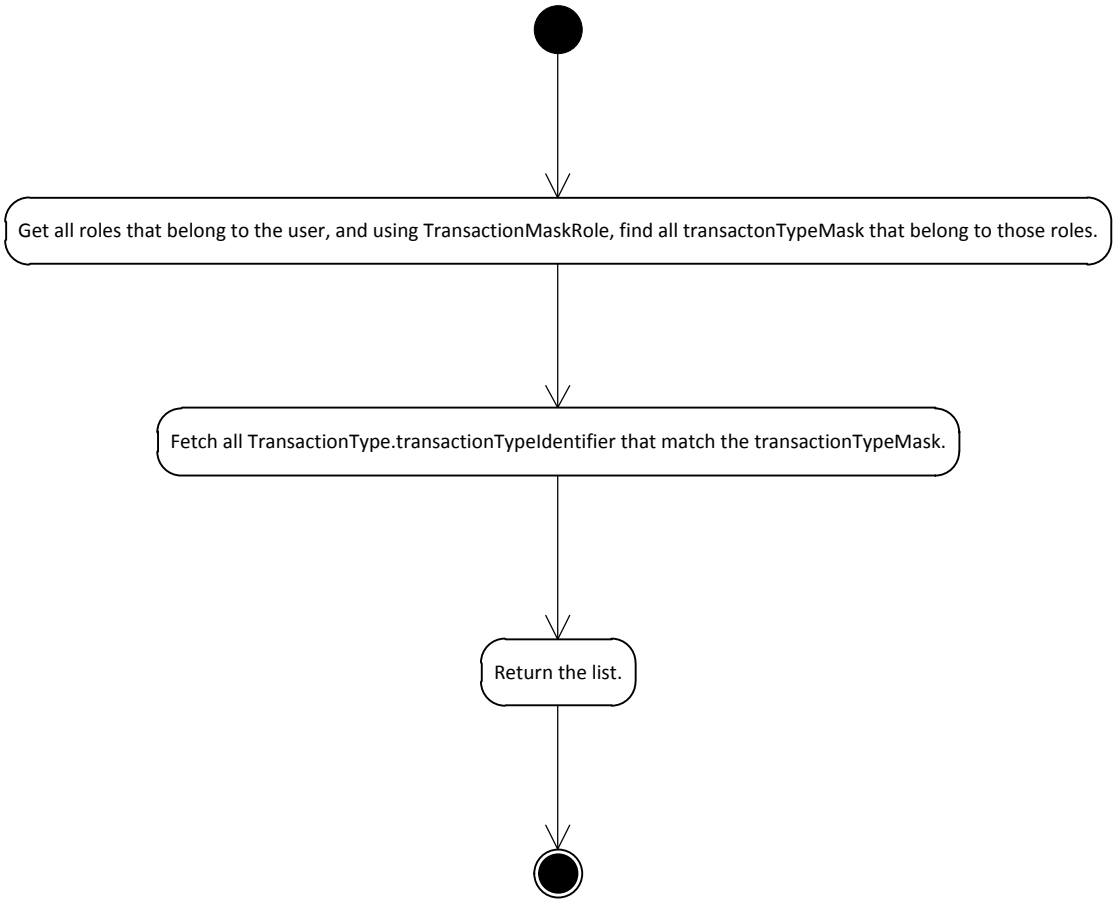
Contents

Access Control (Security Extension) Service [AccessControlService]

The Access Control 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.

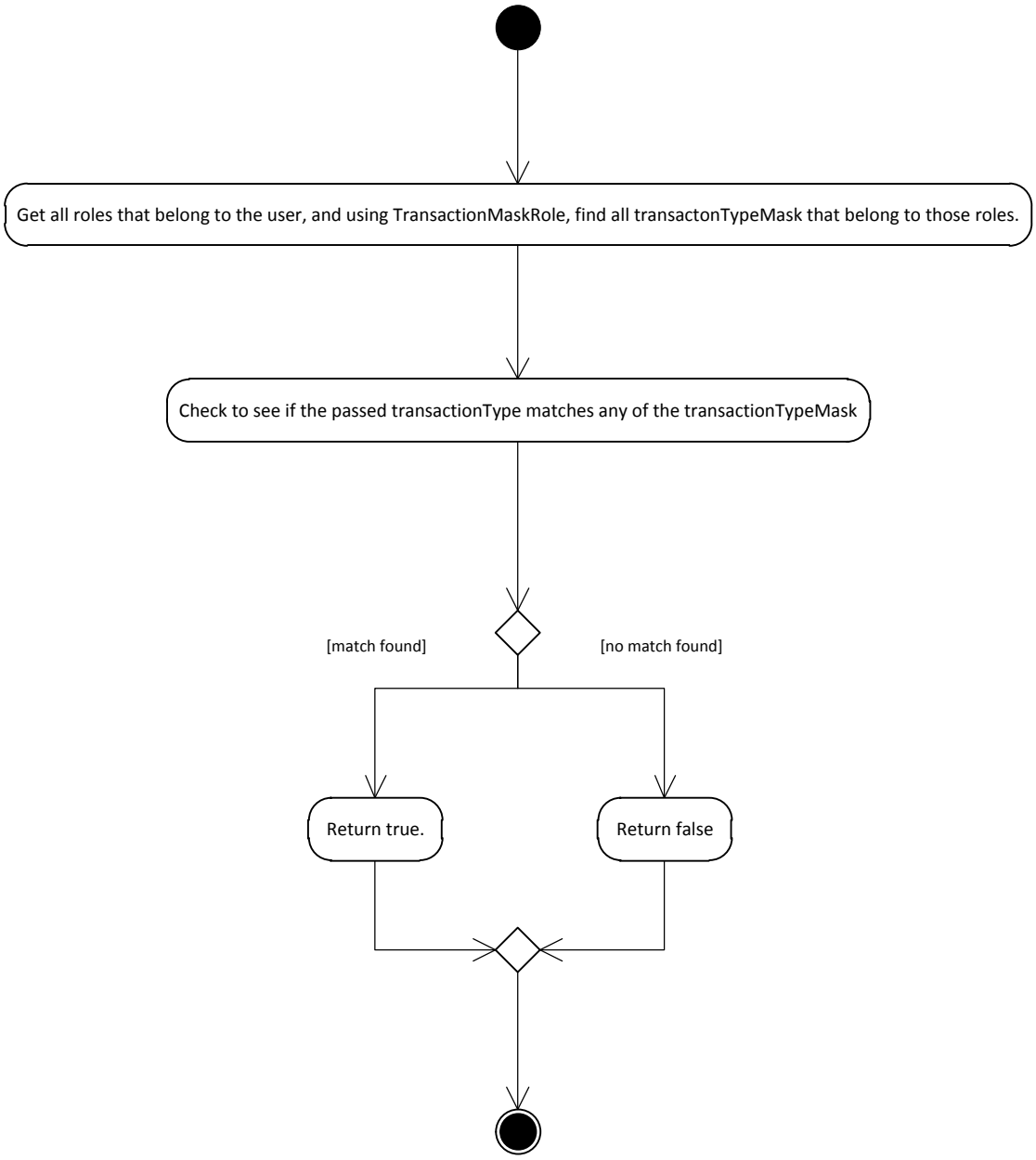
getAllowedTransactionType (String userId)

Returns list of String.



isTransactionTypeAllowed (String userId, String transactionTypeIdentifier)

Returns Boolean.

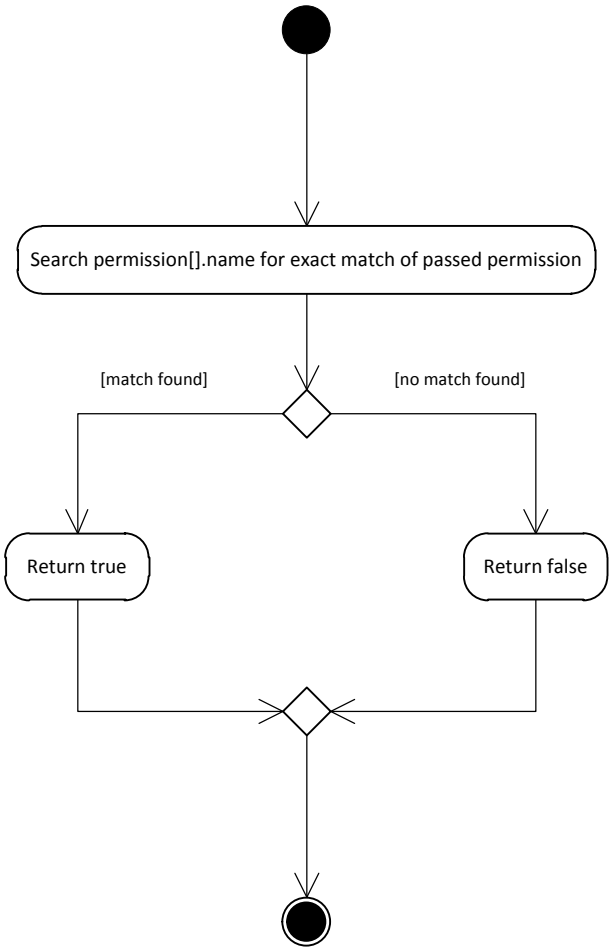


hasPermissions (String userId, String permissionName)

hasPermissions (String userId, String permissionName…)

Returns Boolean.

(formerly isAllowed())



getPermissions(String userId)

Returns list of String.

Return permission associated with the passed user.

getRoles(String userId)

Returns list of String.

Return roles associated with the passed user.

getAllowedTransactionTypeMasks(String userId)

Returns list of String.

Returns a list of transactionTypeMasks associated with a user.

getAllowedTransactionTypes (String userId)

Returns list of String.

Returns a list of allowed transaction types for a user. See getAllowedTransactionTypeMasks().

getTransactionTypesByRoleNames (Set <String> roleNames)

Returns list of String.

Returns list of transaction types accessible by the role(s).

refresh()

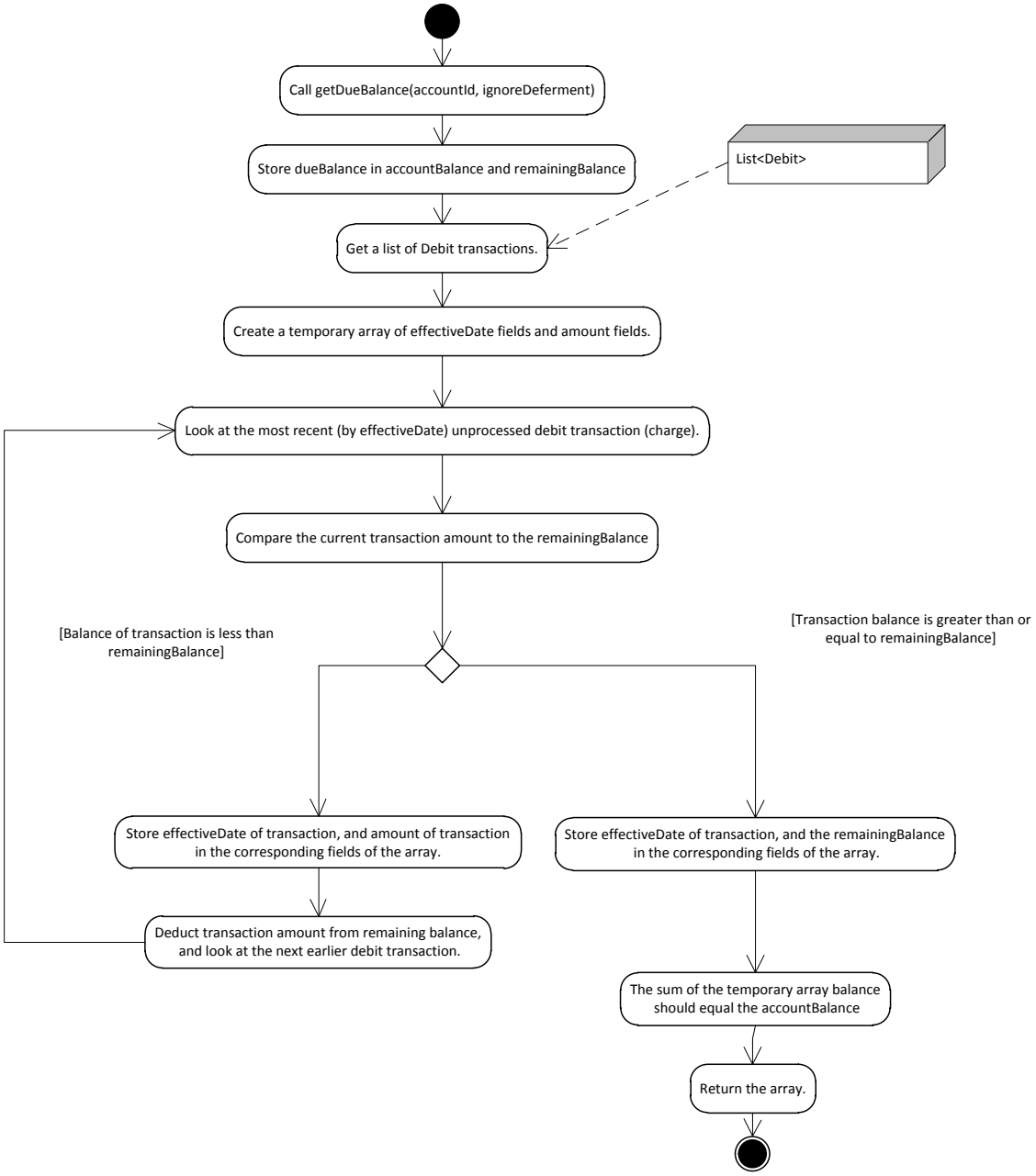
Forces a reload of the cached TransactionTypePermission class. This should be called after changing a permission if it is required that the change be propagated instantly.

Account Service [AccountService]

rebalance(String userId, Boolean ignoreDeferment)

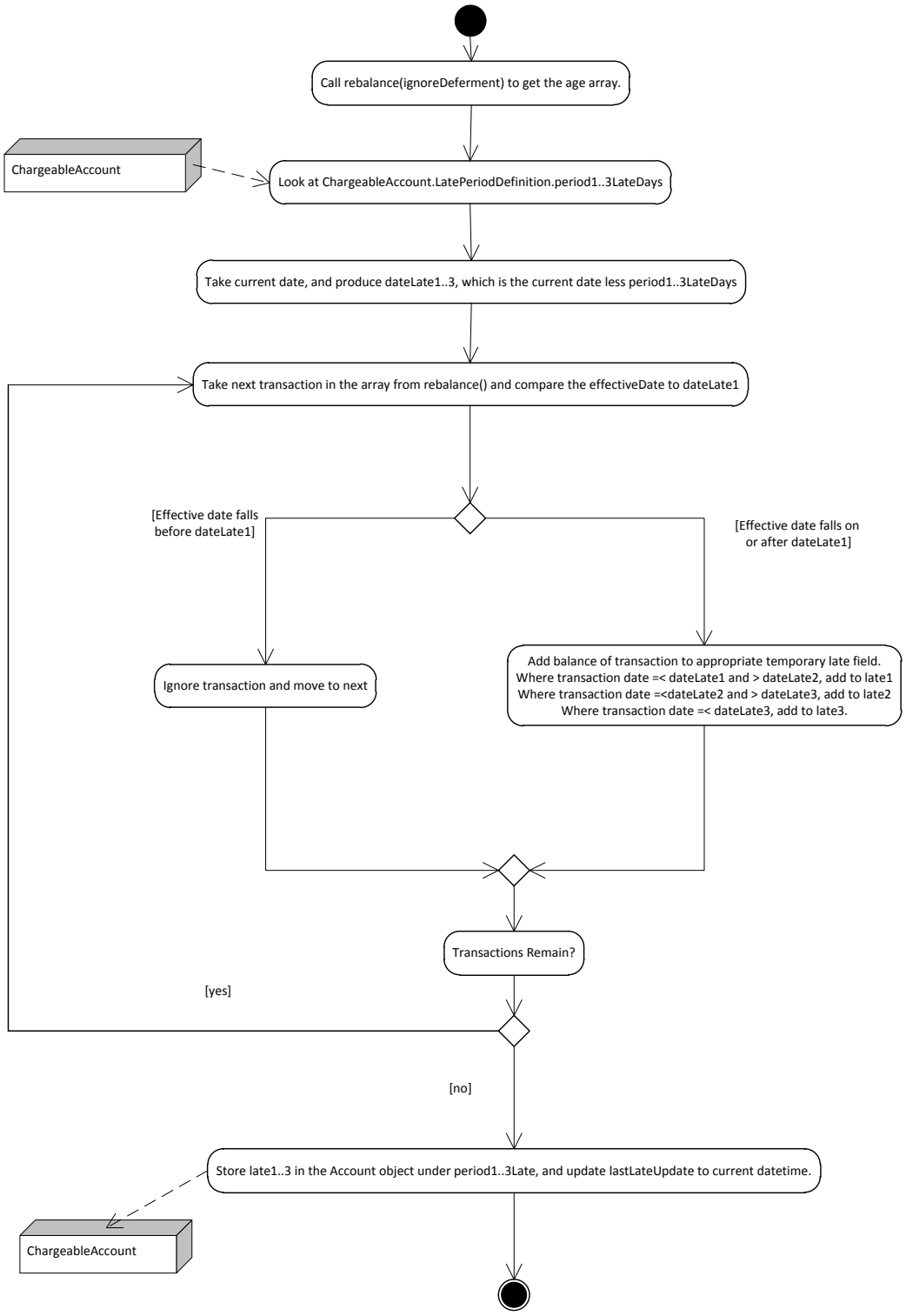
Returns List<Pair<Debit, bigDecimal>>

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.



ageDebt(String userId, Boolean ignoreDeferment)

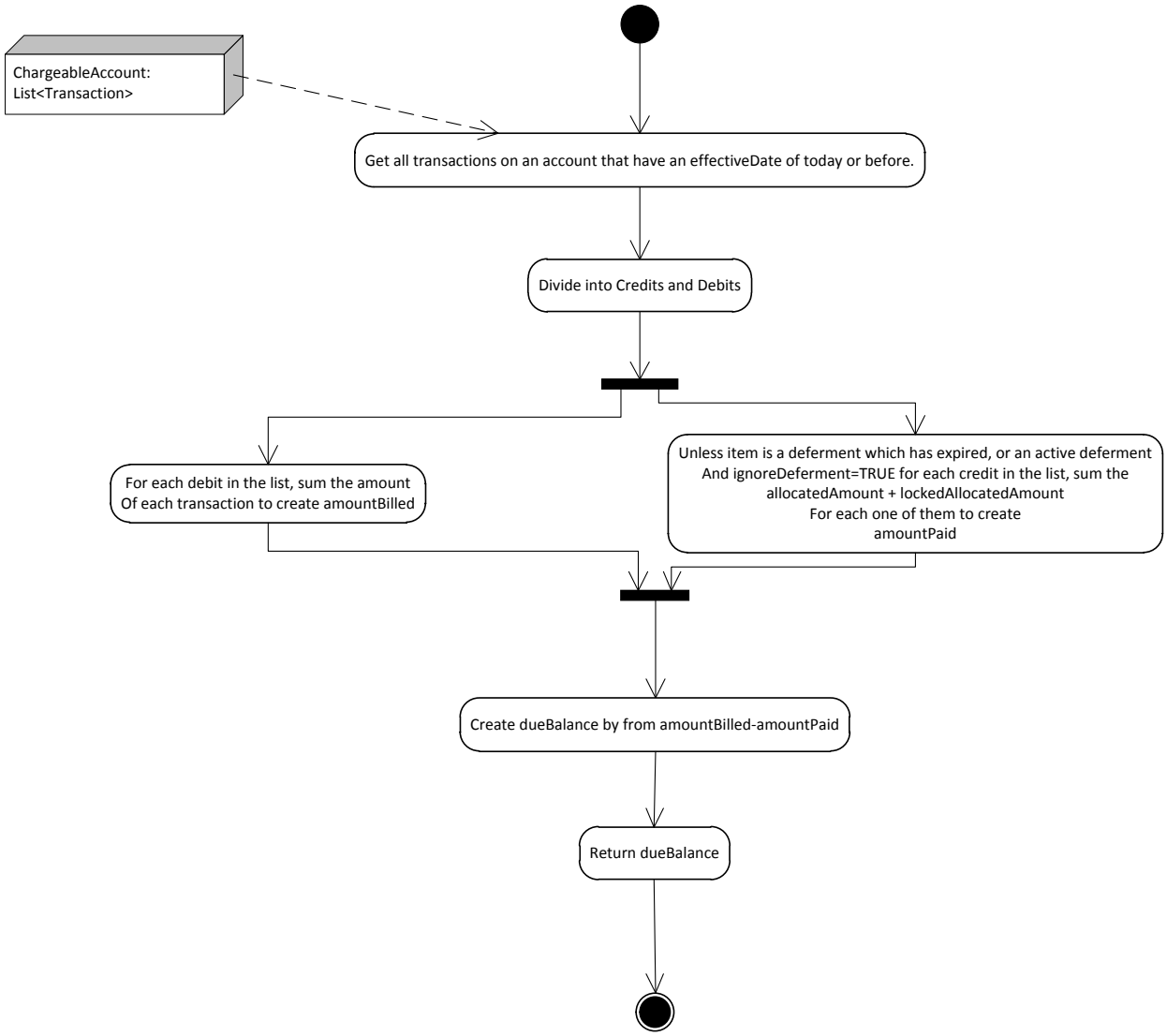
Returns ChargeableAccount.



getDueBalance(String userId, Boolean ignoreDeferment)

Returns BigDecimal.

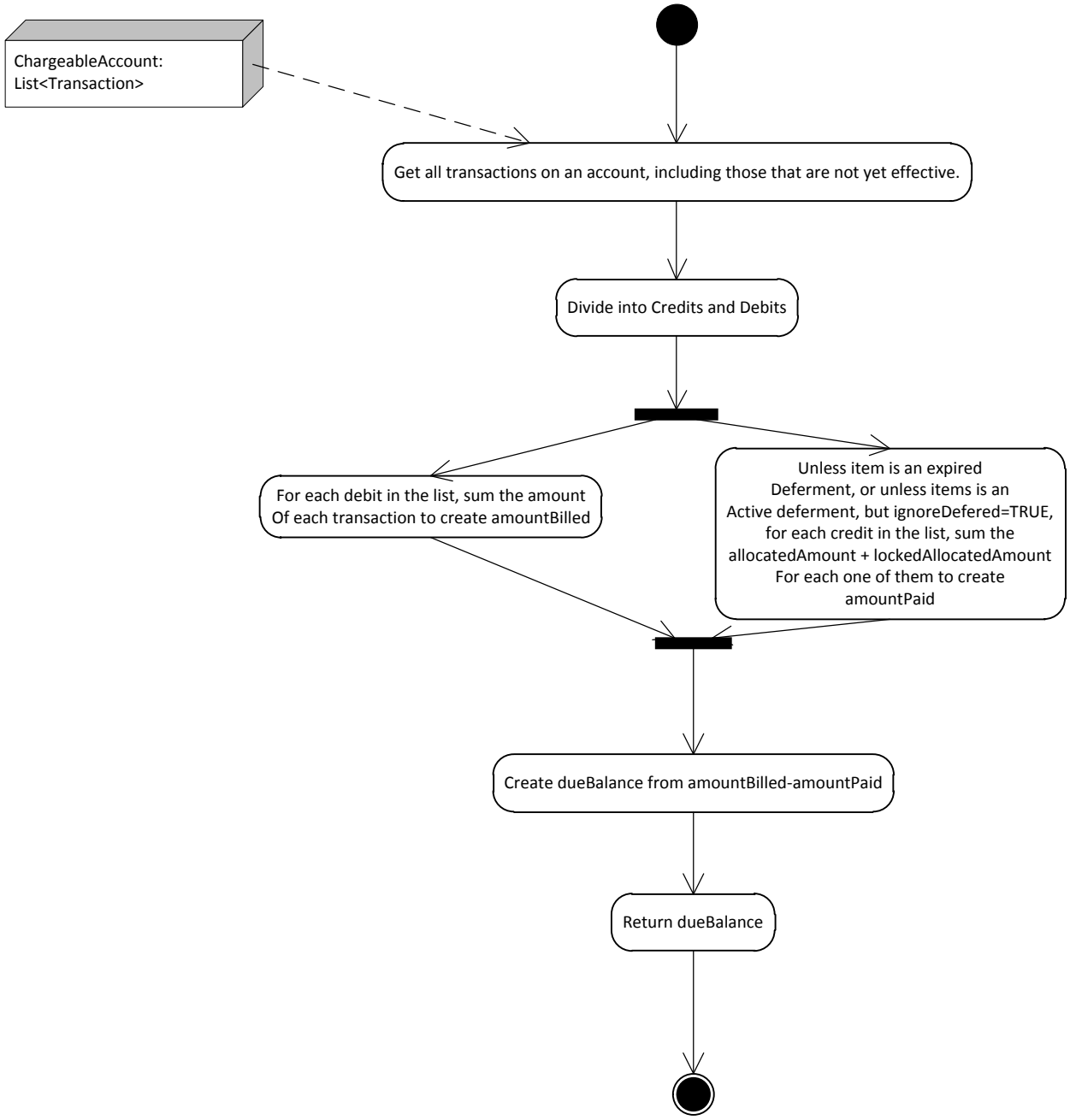
This will get the balance of the account, taking into account only those transactions that are current (effectiveDate is today or before.)



getOutstandingBalance(String userId, Boolean ignoreDeferment)

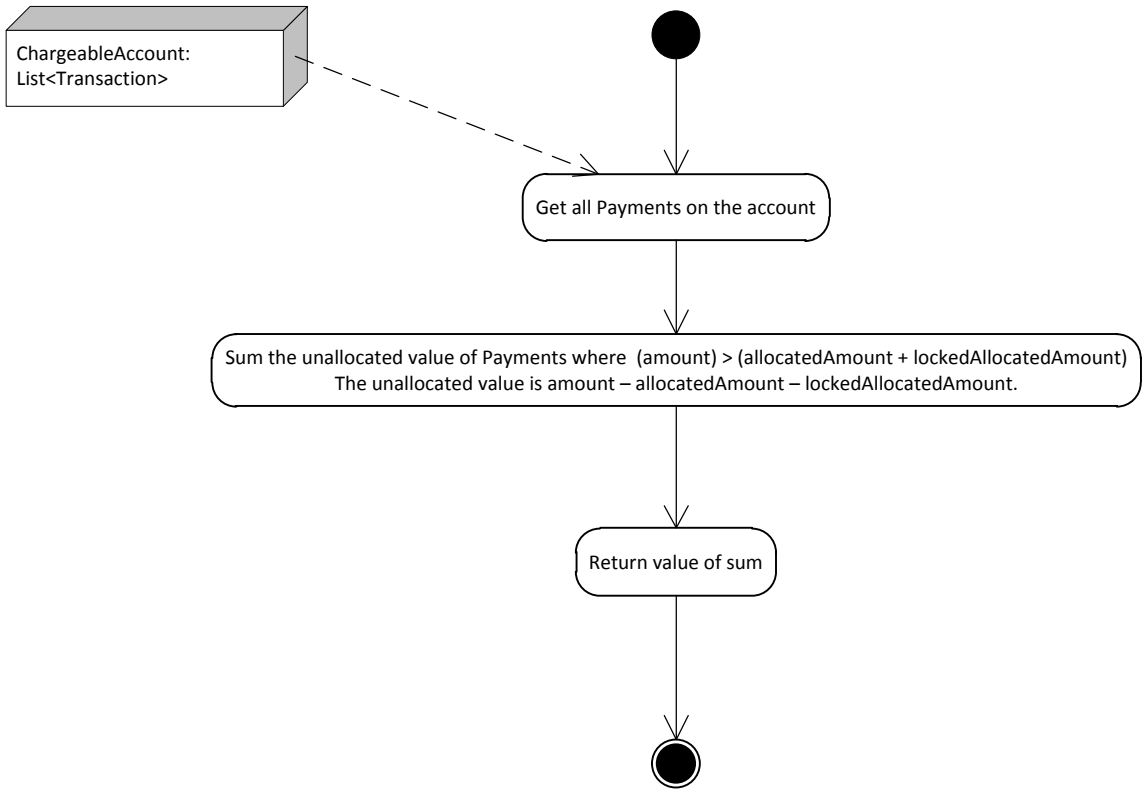
Returns BigDecimal.

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



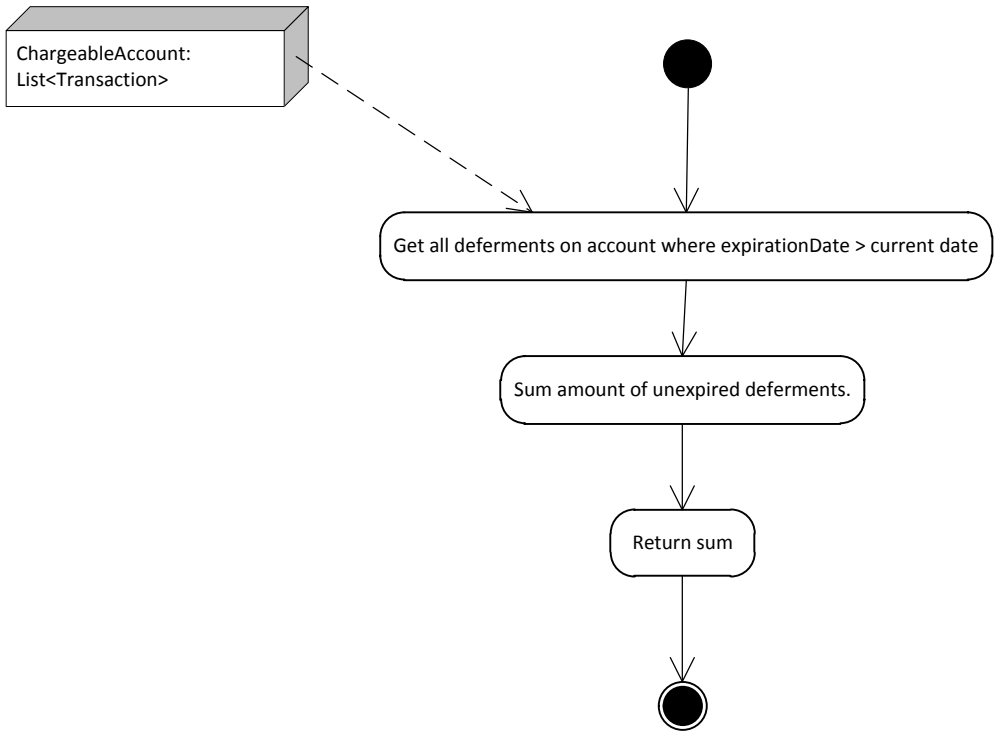
getUnallocatedBalance(String userId)

Returns BigDecimal.



getDeferredAmount(String userId)

Returns BigDecimal.



getFutureBalance (String userId, Boolean ignoreDeferment)

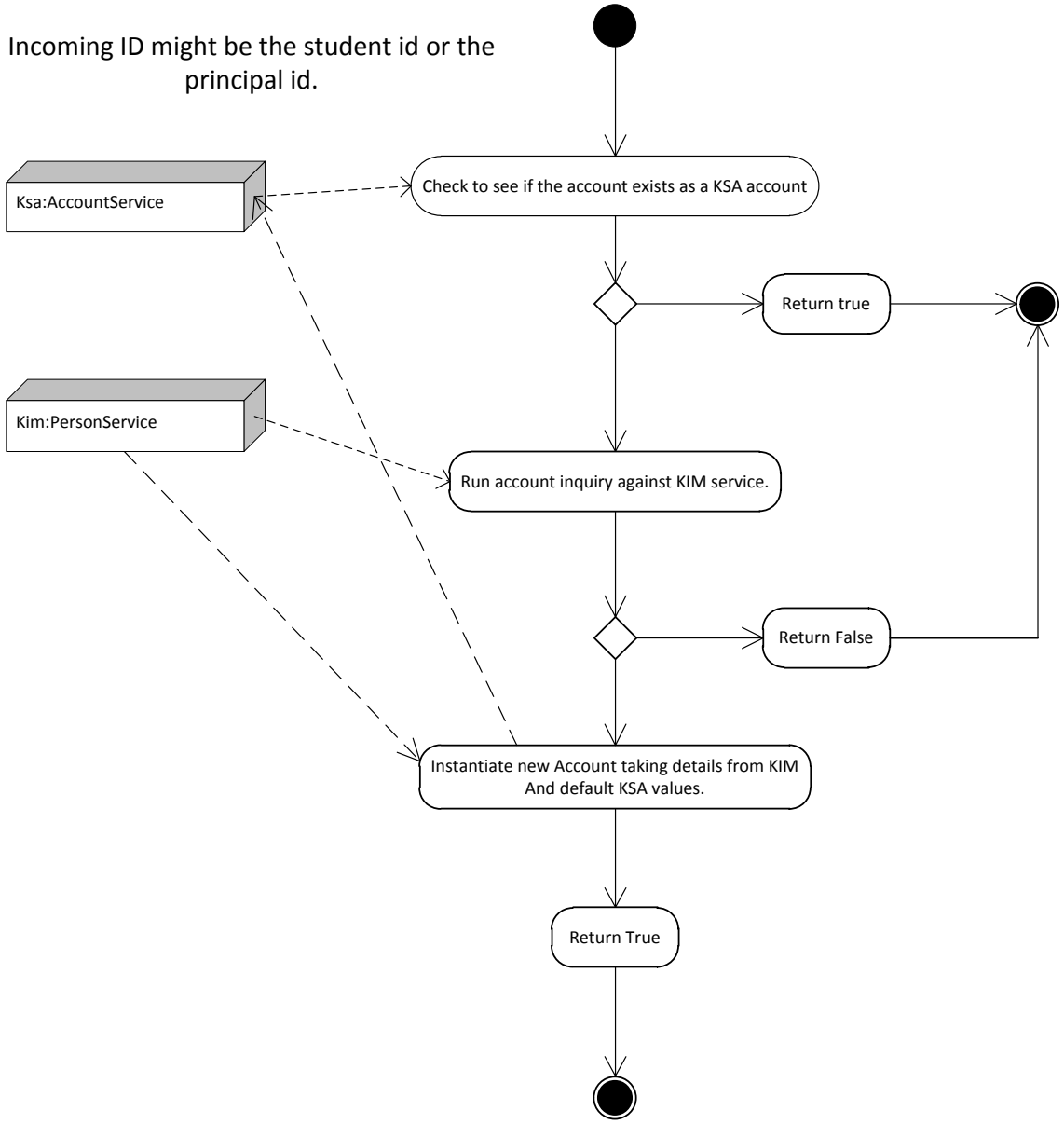
Returns BigDecimal.

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

ksaAccountExists (String userId)

Returns Boolean.

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.



getOrCreateAccount (String userId)

Returns Account.

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 (ksaAccountExists()) 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(String userId)

In progress.

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()

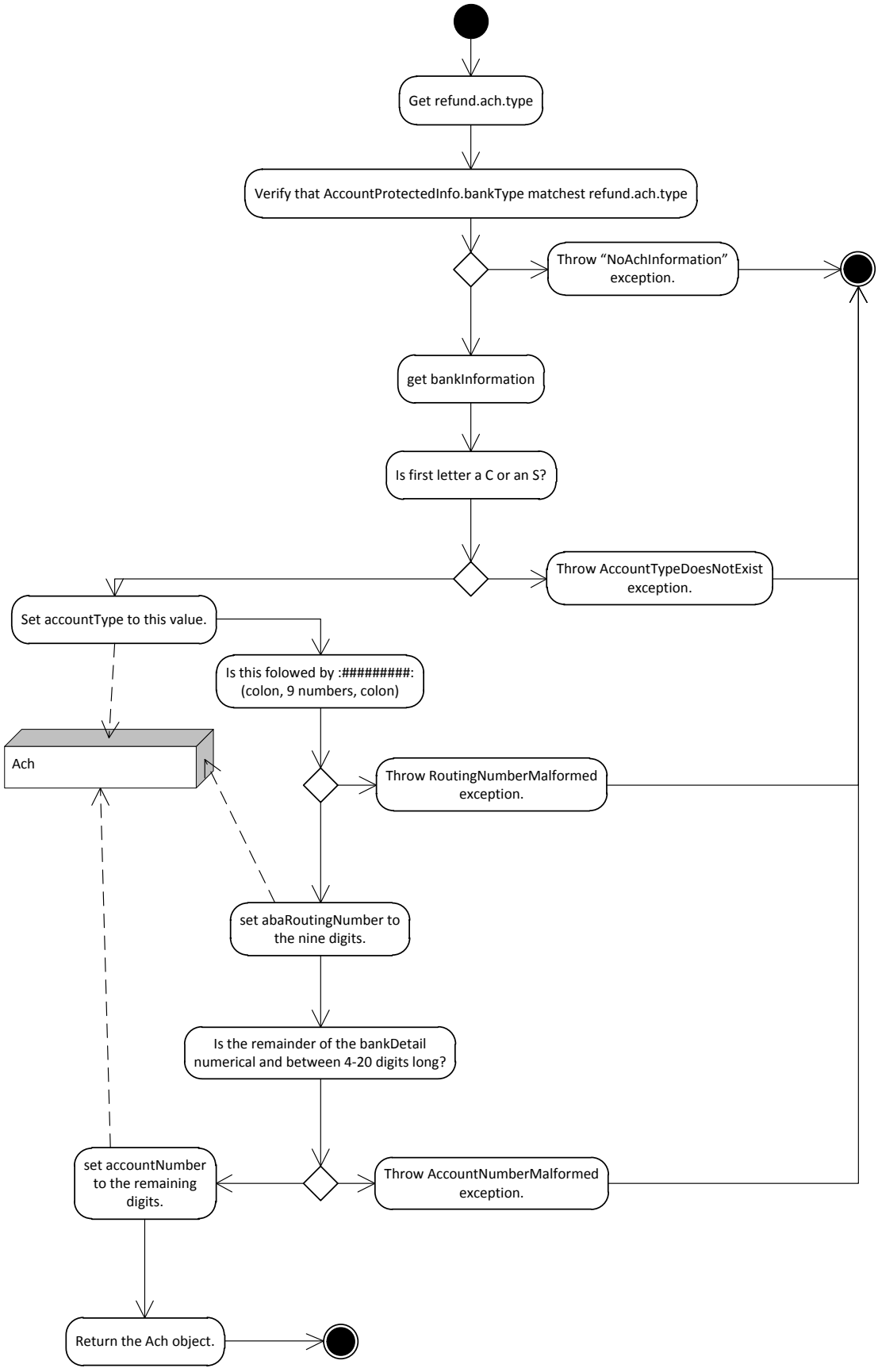
As an example, a school might have a policy that if a student has bounced more than two checks in the last year, they may no longer pass a check to the institution. In this case, a flag would be set each time a bad check were passed to the institution. When getAccountBlockStatus() is called on this account, 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 (String userId)

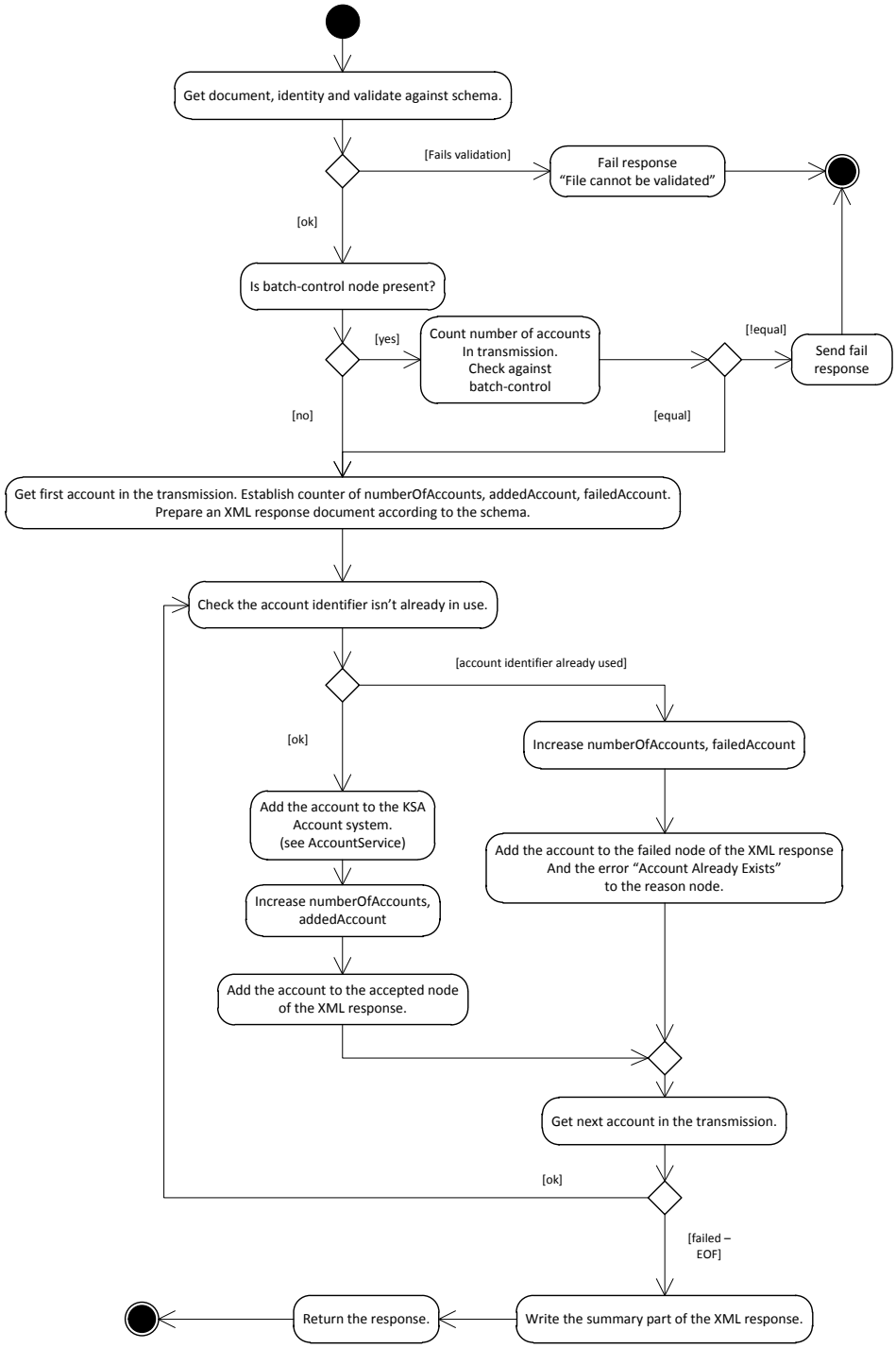
Returns Ach.

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.

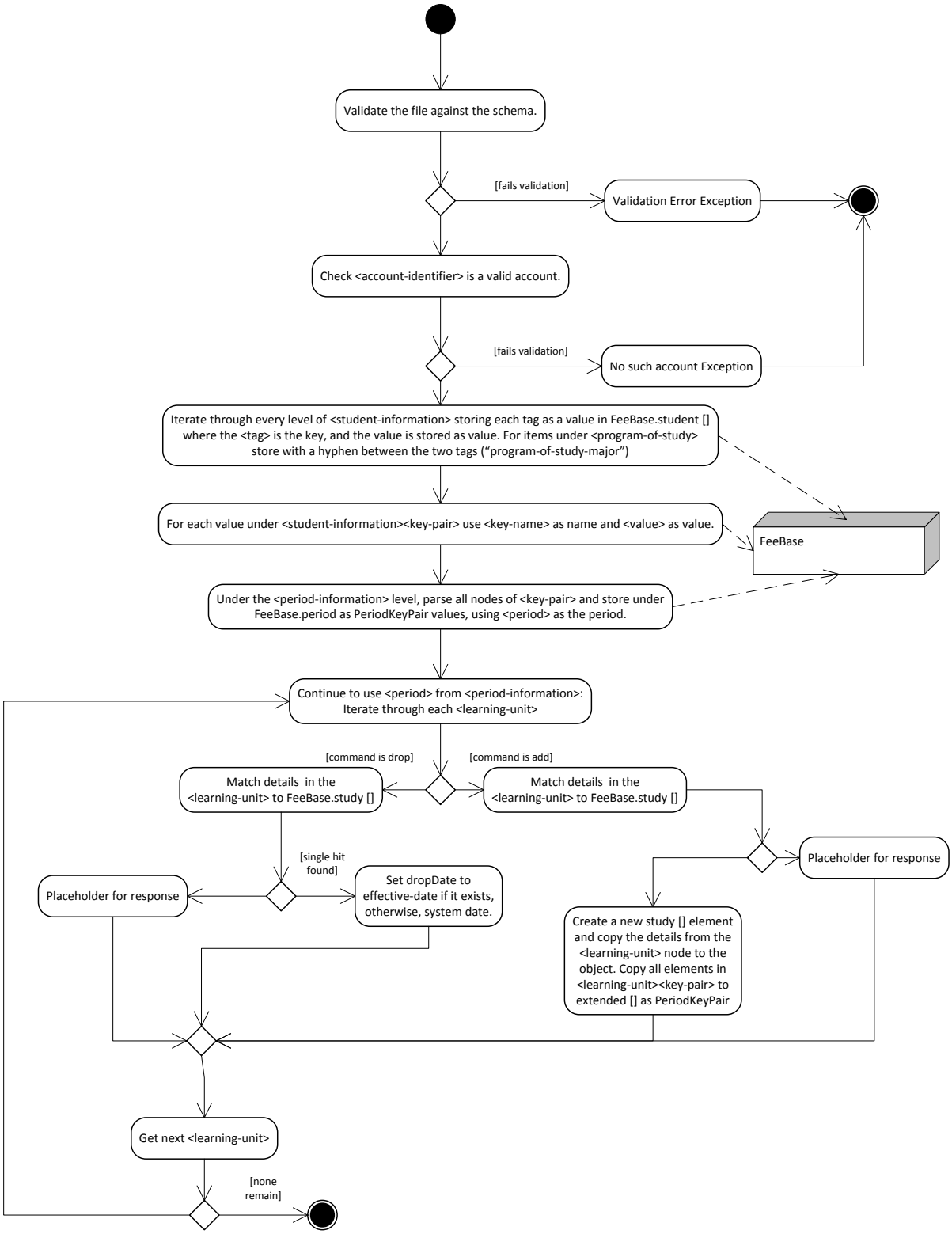
importAccounts (inputFile)

Simple import of accounts via XML.



importStudentProfile (String studentProfile)

Returns FeeBase



writeoffAccount (String accountId)

writeOffAccount (List<String accountId>)

writeoffAccount (String accountId, String memo)

writeOffAccount (List<String accountId>, String memo)

Returns void.

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.

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.

addPersonName (String userId, PersonName personName)

Returns PersonName.

Check ksaAccountExists (userId)

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 (String userId, PostalAddress postalAddress)

Returns PostalAddress.

Check ksaAccountExists(userId)

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 (String userId, ElectronicContact electronicContact)

Returns ElectronicContact.

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.

findAccountsByNamePattern (String namePattern)

Returns List <Account>

Return a list of accounts matching the name pattern passed.

getFullAccounts ()

Returns List <Account>

Gets all KSA asccounts and their associations.

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.

Activity Service [ActivityService]

The Activity Service controls the recording of activity in the system.

getActivity (Long id)

Returns Activity.

Return the activity with the associated id.

getActivities ()

getActivities (String userId)

Returns List <Activity>

Returns a list of all activities.

persistActivity (Activity activity)

Returns Long, the identifier for the activity.

Persists the activity in the database.

deleteActivity (Long id)

Returns boolean.

Removes the associated activity in the database, and return true.

Currency Service

The currency service is responsible for currency establishment and tracking within the system.

getCurrency (Long id)

getCurrency (String code)

Returns Currency.

Get the currency identified and return it.

getCurrencies()

Returns List <Currency>

Returns list of all currencies in the system.

persistCurrency (Currency currency)

Returns Long id, identifier for the currency.

Persists the passed currency in the database, and return its identifier.

deleteCurrency (Long id)

Returns Boolean.

Remove the currency identified, and return true.

Fee Management Service (Proof of Concept)

calculateFees (feeBase, period)

Calls the fee assessment rules, passing the feeBase object for the student in question.

calcluateChargeByCreditToMax (int numberOfCredits, BigDecimal amountPerCredit, BigDecimal maximumAmount)

Returns BigDecimal.

Multiply amountPerCredit by numberOfCredits, and return the smaller of the result or maximumAmount, unless maximumAmount = -1, in which case, return only the value of the multiplication.

getFeeBase (String userId)

Returns FeeBase.

Return the feeBase object for a given account.

createKeyPair (FeeBase feeBase, String name, String value)

createKeyPair (FeBase feeBase, String name, String value, LearningPeriod period)

createKeyPair (LearningUnit learningUnit, String name, String value)

Returns newly created KeyPair/PeriodKeyPair.

Under feebase, create a new key pair with the name and value passed. If the key pair already exists, overwrite it. If period is passed, create a periodKeyPair.

getKeyPairValue (FeeBase feeBase, String name)

getKeyPairValue (FeeBase feeBase, String name, LearningPeriod period)

getKeyPairValue (LearningUnit learningUnit, name)

Returns String value.

Return the value of the key pair referenced, return null if there is no key pair with that name.

removeKeyPair (FeeBase feeBase, String name)

removeKeyPair (FeeBase feeBase, String name, LearningPeriod learningPeriod)

removeKeyPair (LearningUnit learningUnit, String name)

Returns void.

If the key pair exists, remove it.

updateKeyPair(FeeBase feeBase, String name, String value)

updateKeyPair(FeeBase feeBase, String name, String value, LearningPeriod period)

updateKeyPair(LearningUnit learningUnit, String name, String value)

Return void.

If the keypair already exists, alter it to the new value.

containsKeyPair (FeeBase feeBase, String name)

containsKeyPair (LearningUnit learningUnit, String name)

Returns Boolean.

Returns true if FeeBase / LearningUnit contains a KeyPair or subtype with the given name. Otherwise returns false.

saveLearningUnit (LearningUnit learningUnit)

Returns void.

Perist the learningUnit.

findLearningPeriods (Date dateFrom, Date dateTo)

Returns List <LearningPeriod>

Returns a list of learning periods that are within the date range passed.

getCurrentPeriod()

Returns LearningPeriod.

Returns the period that the rules are currently working on.

getStudentData (String userId)

Returns List <KeyPair>

Returns all keypairs associated with an account.

getLearningPeriodData (String userId)

Returns List <PeriodKeyPair>

For a given userId, return the account's PeriodLearningKeys.

getStudy (String userId)

Return List <LearningUnit>

For a given account, return a list of the LearningUnit associated with it.

General Ledger Service [GeneralLedgerService]

addGeneralLedgerTransaction(Long transactionIdentifier, String generalLedgerAccount, BigDecimal amount, GlOperationType glOperation, String generatedInformation)

addGeneralLedgerTransaction(Long transactionIdentifier, Strnig generalLedgerAccount, BigDecimal amount, GlOperationType glOperation, Strng generatedInformation, Boolean isQueued)

Returns GlTransaction.

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.

getGeneralLedgerType (String glTypeCode)

Returns GeneralLedgerType

getDefaultGeneralLedgerMode ()

Returns GeneralLedgerMode.

getDefaultGeneralLedgerType ()

Returns GeneralLedgeType.

isGlAccountValid (String glAccount)

Returns Boolean.

This is a service that is designed to be overridden. In the standard release, this will check the account with via a simple internal lookup. This is expected to change when KFS supports some level of remote account lookup.

searchForGeneralLedgerAccounts (String generalLedgerAccount)

Returns List <TransactionType>.

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

createGeneralLedgerType (String code, String name, String description, String generalLedgerAssetAccount, GlOperation glOperationOnCharge)

Returns GeneralLedgerType.

Check that no generalLedgerType with the same code exists, otherwise thrown an exception.

Call isGeneralLedgerAccountValid () on the account. If this fails, throw an exception.

Create a new GeneralLedgerType object with the parameters passed.

Set creatorId to the current user, and creationDate to the current date/time.

editGeneralLedgerType (GeneralLedgerType generalLedgerType, String name, String description, String generalLedgerAssetAccount, GlOperation generalLedgerOperationOnCharge)

Returns GeneralLedgerType.

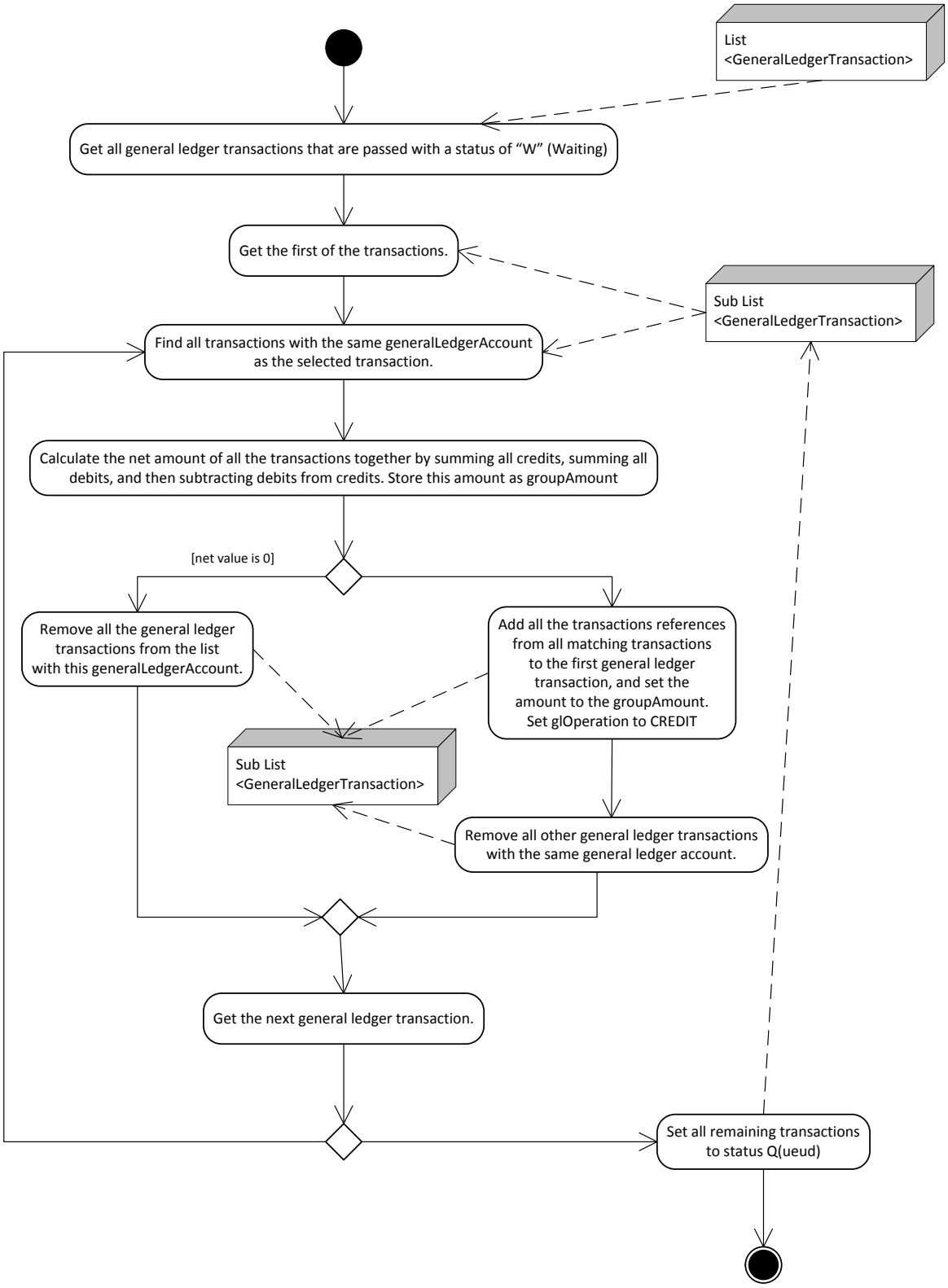
Call isGeneralLedgerAccountValid () on the account. If this fails, throw an exception.

Update the generalLedgerType with the values.

Set editorId to the current user, and lastUpdated to the current date/time.

summarizeGeneralLedgerTransactions (list<GlTransaction> glTransactions)

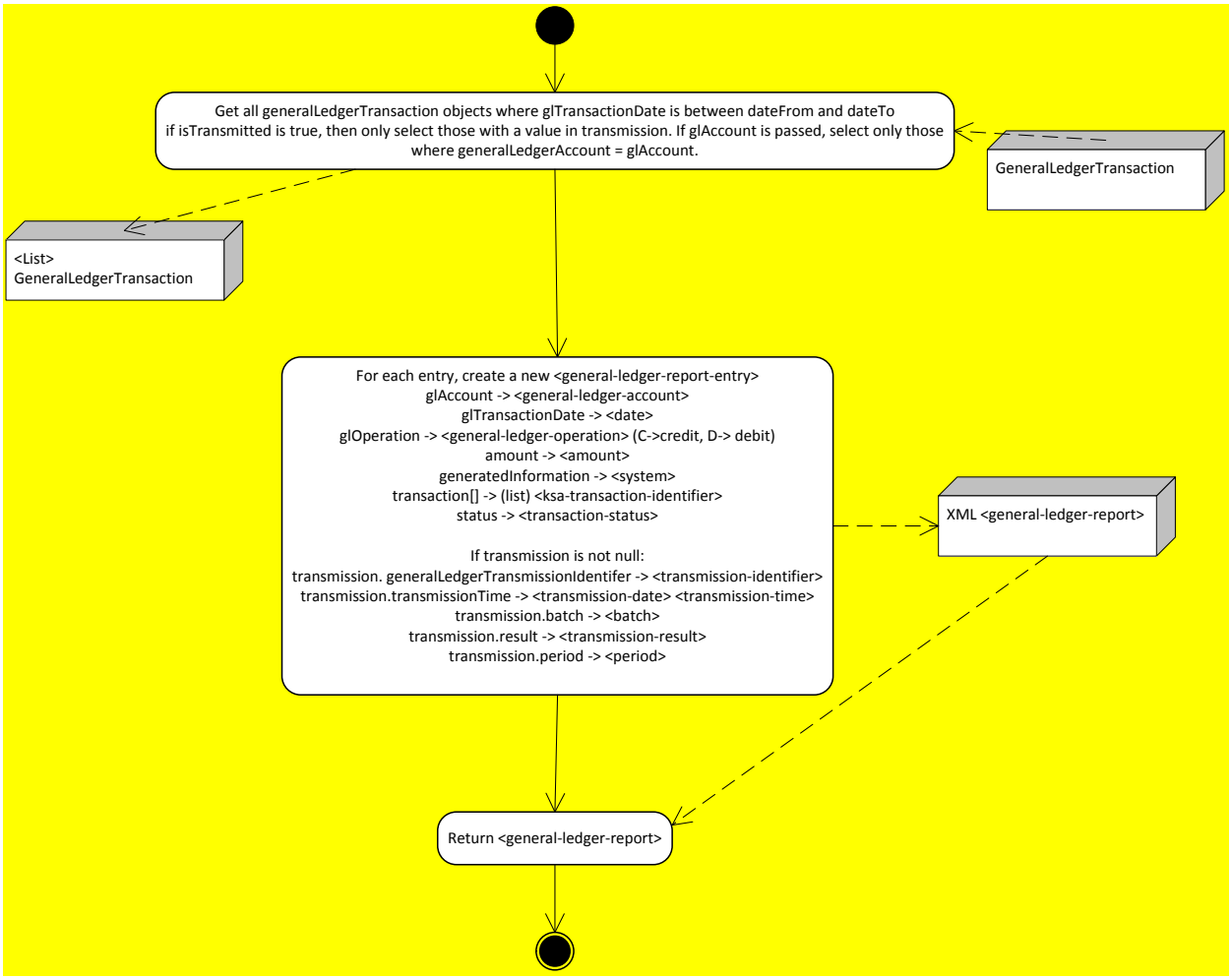
Returns void.


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 all credits, summing all debits, and then subtracting debits from credits. Store this amount as groupAmount
[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 groupAmount. Set glOperation to CREDIT
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.



synchronized prepareGlTransmission()

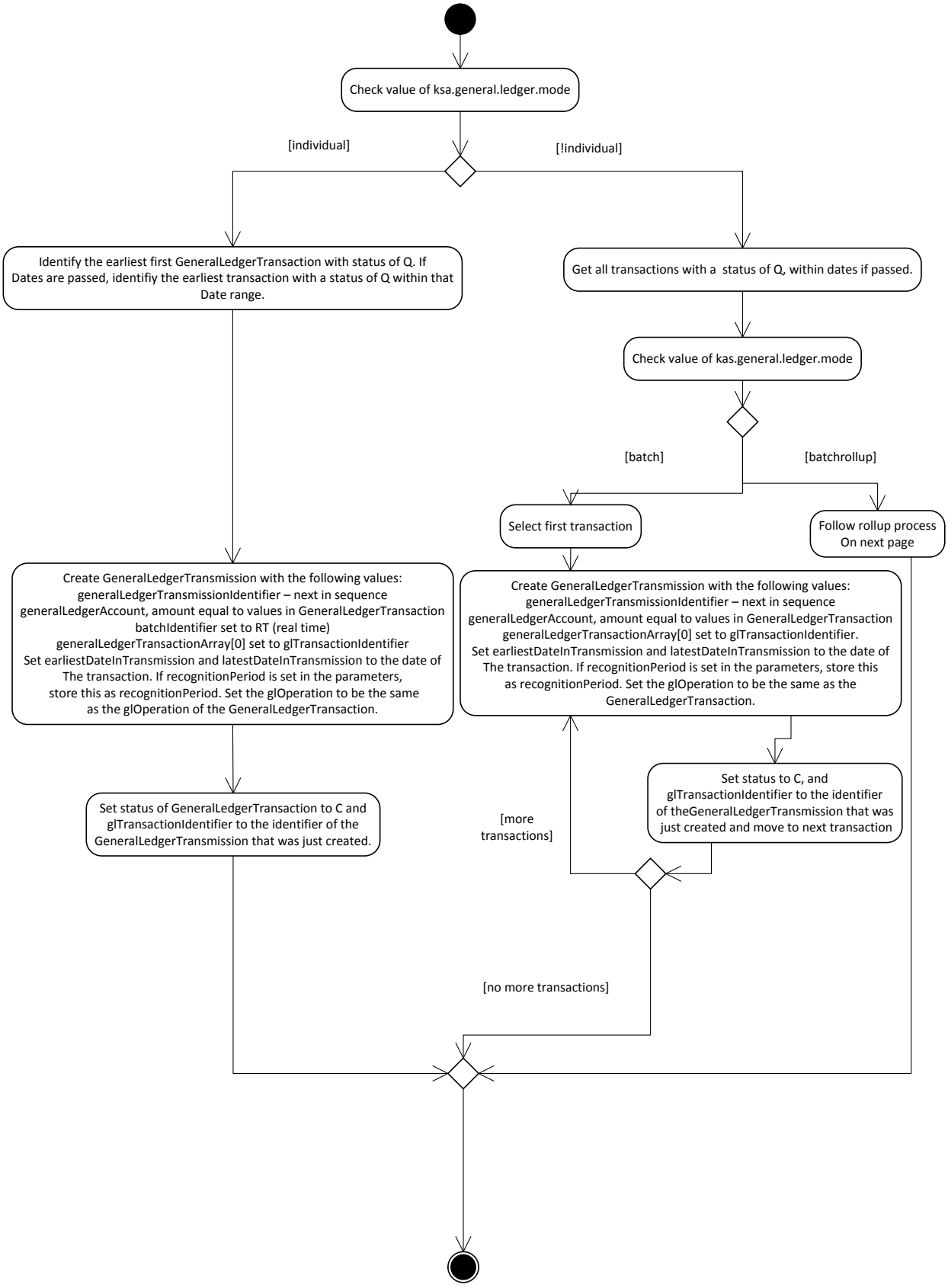
synchronized prepareGeneralLedgerTransmission (Date effectiveDateFrom, Date effectiveDateTo, String recognitionPeriod)

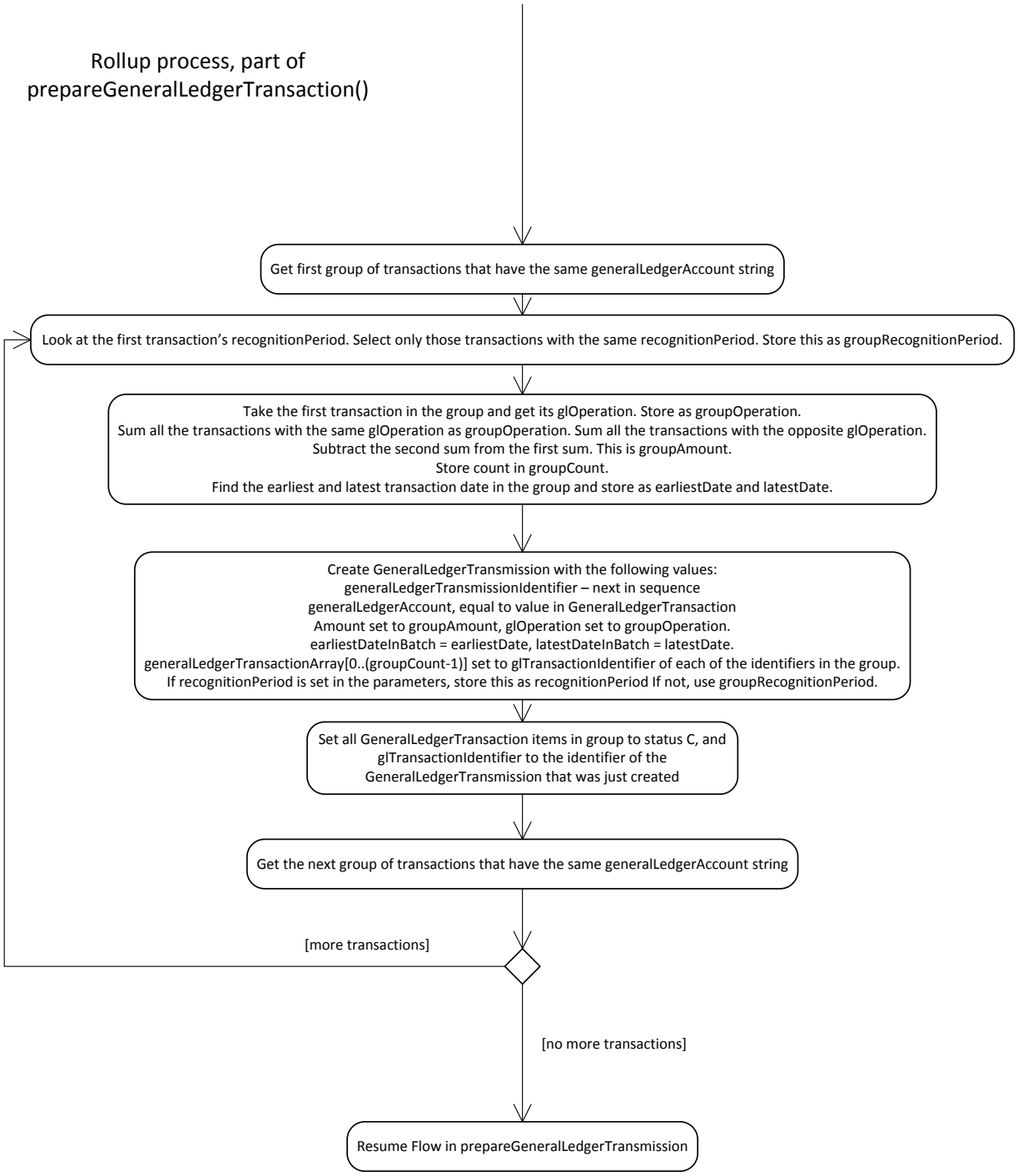
synchronized prepareGlTransmissionByRecognitionDate (Date recognitionDateFrom, Date recognitionDateTo, String recognitionPeriod)

Returns void.

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.

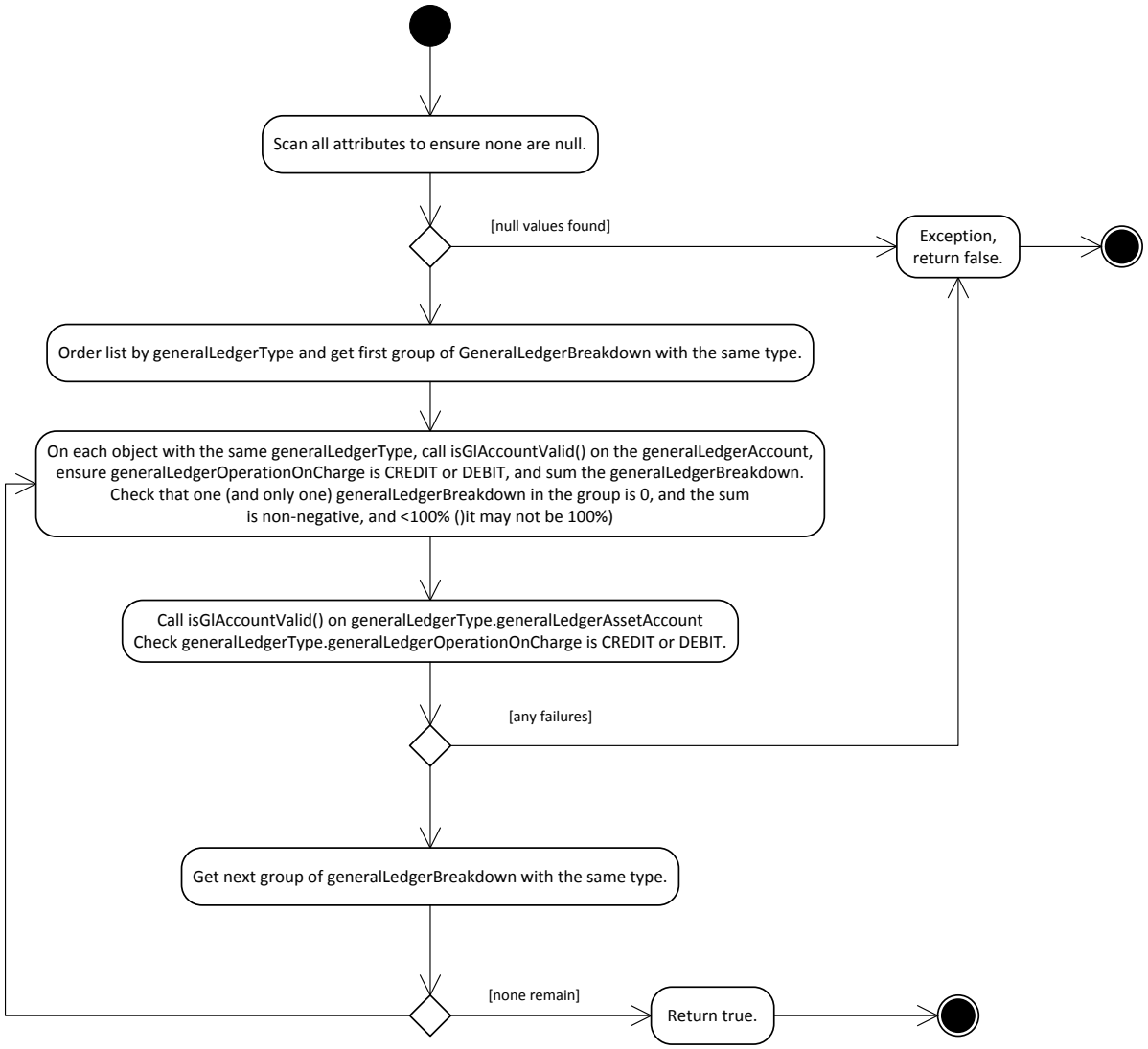

Check value of ksa.general.ledger.mode
[individual]
[!individual]
Identify the earliest first GeneralLedgerTransaction with status of Q. If Dates are passed, identifiy the earliest transaction with a status of Q within that Date range.
Get all transactions with a status of Q, within dates if passed.
Check value of kas.general.ledger.mode
[batch]
[batchrollup]
Select first transaction
Follow rollup process On next page
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. Set the glOperation to be the same as the glOperation of the GeneralLedgerTransaction.
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 the glOperation to be the same as the GeneralLedgerTransaction.
Set status of GeneralLedgerTransaction to C and glTransactionIdentifier to the identifier of the GeneralLedgerTransmission that was just created.
Set status to C, and glTransactionIdentifier to the identifier of theGeneralLedgerTransmission that was just created and move to next transaction
[more transactions]
[no more transactions]


Rollup process, part of
prepareGeneralLedgerTransaction()
Get first group of transactions that have the same generalLedgerAccount string
Look at the first transaction's recognitionPeriod. Select only those transactions with the same recognitionPeriod. Store this as groupRecognitionPeriod.
Take the first transaction in the group and get its glOperation. Store as groupOperation.
Sum all the transactions with the same glOperation as groupOperation. Sum all the transactions with the opposite glOperation.
Subtract the second sum from the first sum. This is 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, glOperation set to groupOperation.
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 If not, use groupRecognitionPeriod.
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

isGeneralLedgerBreakdownValid (List<GeneralLedgerBreakdown generalLedgerBreakdown>)

Returns Boolean.

Validates the details within the general leger breakdown block and ensures that it can be used to break down a transaction.



createGeneralLedgerType (String code, String name, String description, String assetAccount, GeneralLedgerOperation operationOnCharge)

Returns GeneralLedgerType

Ensure that no other GeneralLedgerType with the same code exists.

Do isGlAccountValid on the assetAccount.

Ensure operationOnCharge is CREDIT or DEBIT.

If any of these fail, throw exception.

Otherwise, create a new generalLedgerType object with the included attributes, set id to the next identifier.

persistGeneralLedgerType (GeneralLedgerType generalLedgerType)

Returns GeneralLedgerType

Check no other generalLedgerType with the same code exists.

Do isGlAccountValid on the assetAccount.

Ensure operationOnCharge is CREDIT or DEBIT.

If any of these fail, throw exception.

If creatorId is null, set to current user, and set creationDate to current date/time

Otherwise, set editorId to current user and set lastUpdate to current date/time.

Persist the object.

General Ledger Transmission Service – Part of General Ledger Service

These methods are implementation specific, and will need to be modified depending on the GL numbering system used by the school, as well as the general ledger in use. By default, KSA can produce KFS transactions. These reference processes show how the information from KSA is used to produce a list of XML transactions to be sent to KFS at the University of Maryland, College Park.

transmitToGeneralLedger ()

Generic service designed to be overridden at the institution.

For the purpose of the proof of concept that will operate with Maryland's system -> KFS as a file upload:

Get the next General Leger Batch number

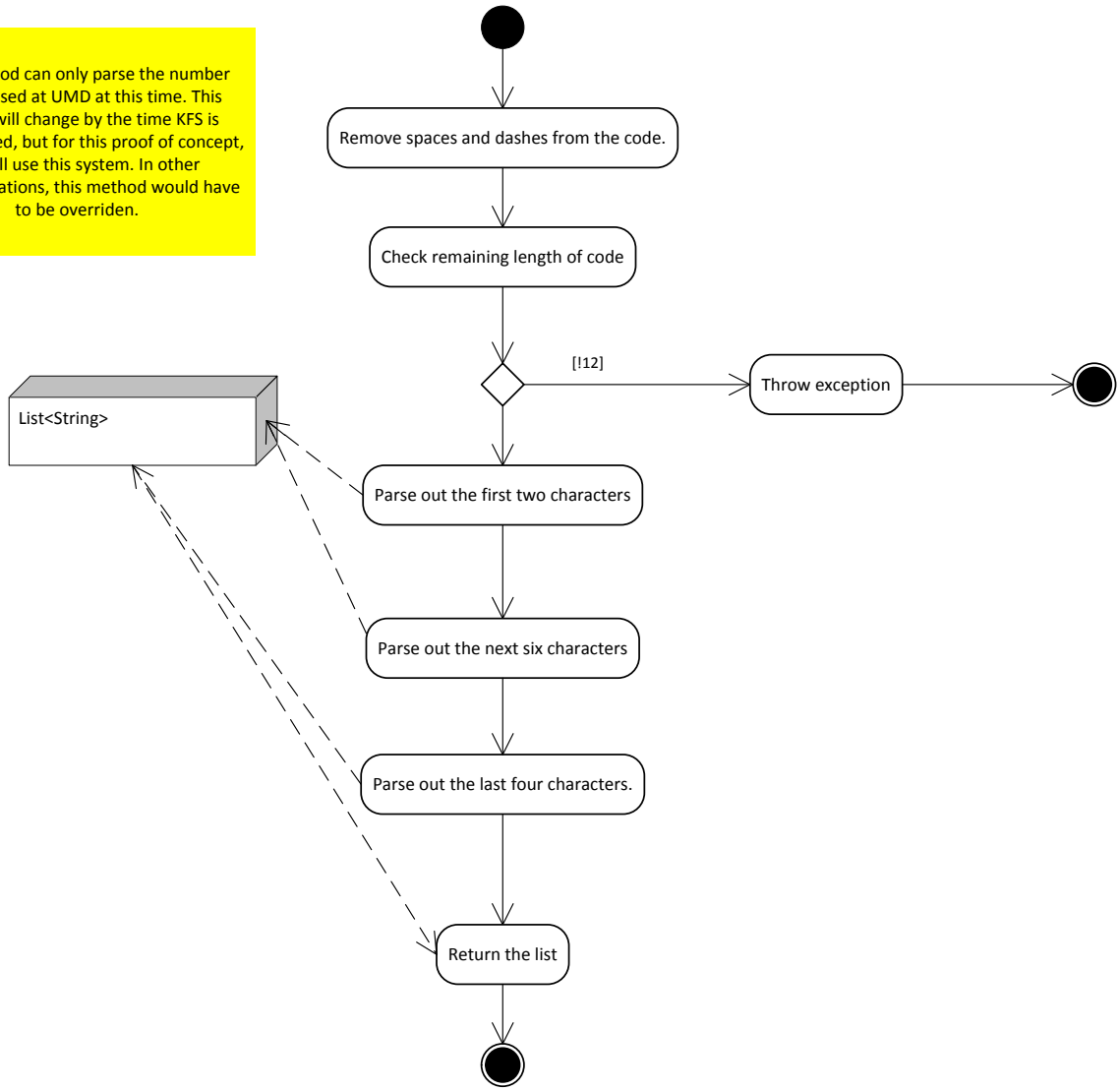
Get all the GeneralLedgerTransmission objects with a result that is blank, and pass to

generateGeneralLedgerTransmission () passing the list and the batch number just gained.

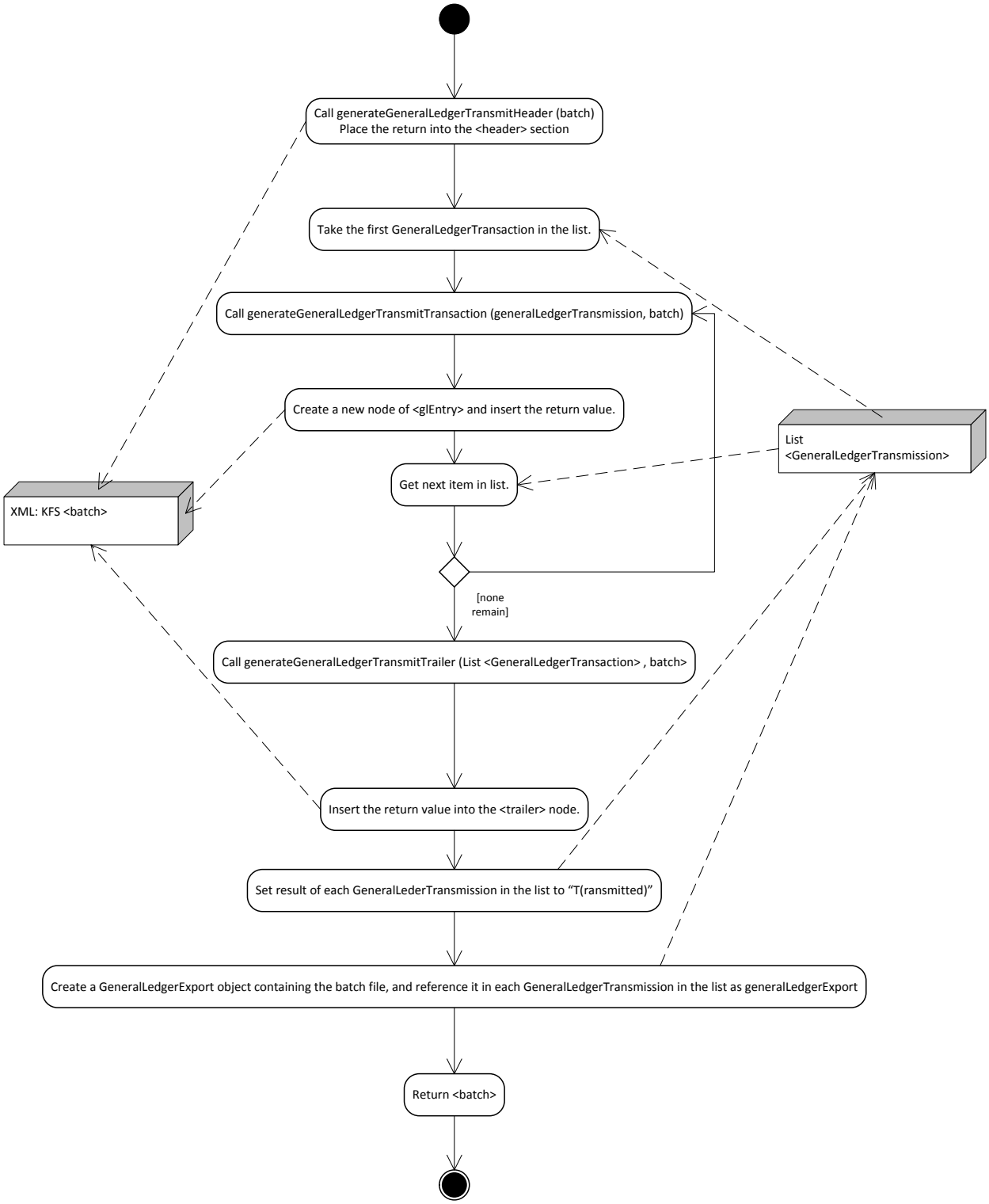
Return the completed XML file to the user to be uploaded to KFS.

parseGeneralLedgerAccountNumber (generalLedgerAccount)

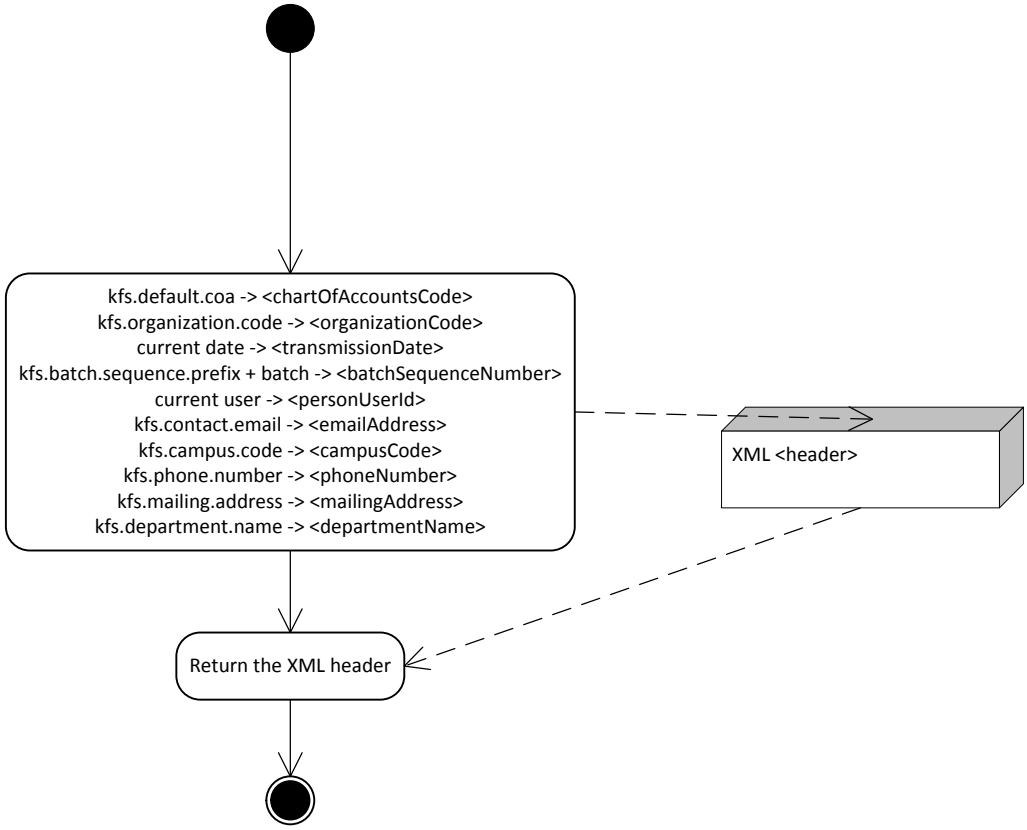
This method can only parse the number system used at UMD at this time. This system will change by the time KFS is implemented, but for this proof of concept, we will use this system. In other implementations, this method would have to be overriden.



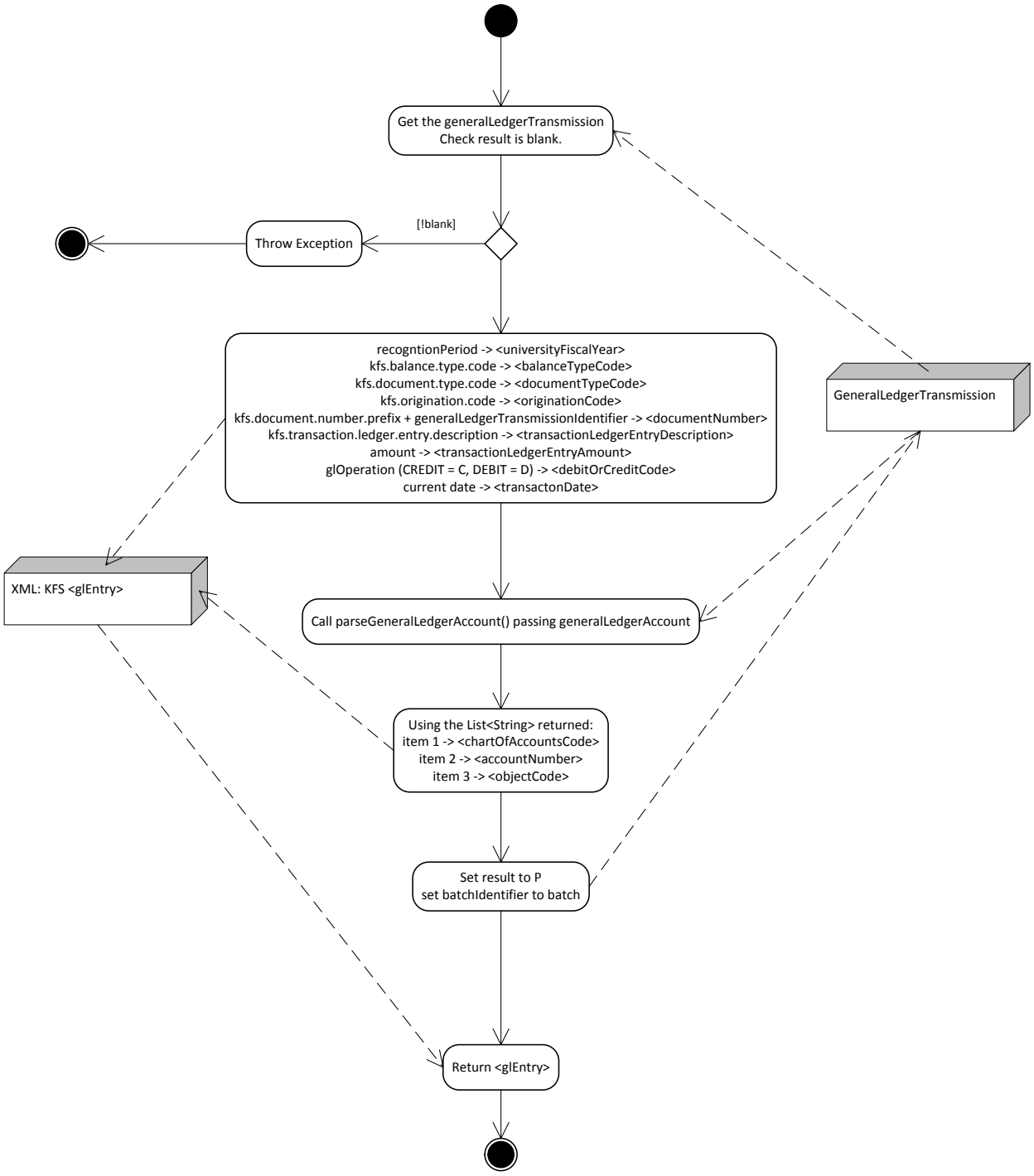
generateGeneralLedgerTransmission (list <GeneralLedgerTransmission>, batch)



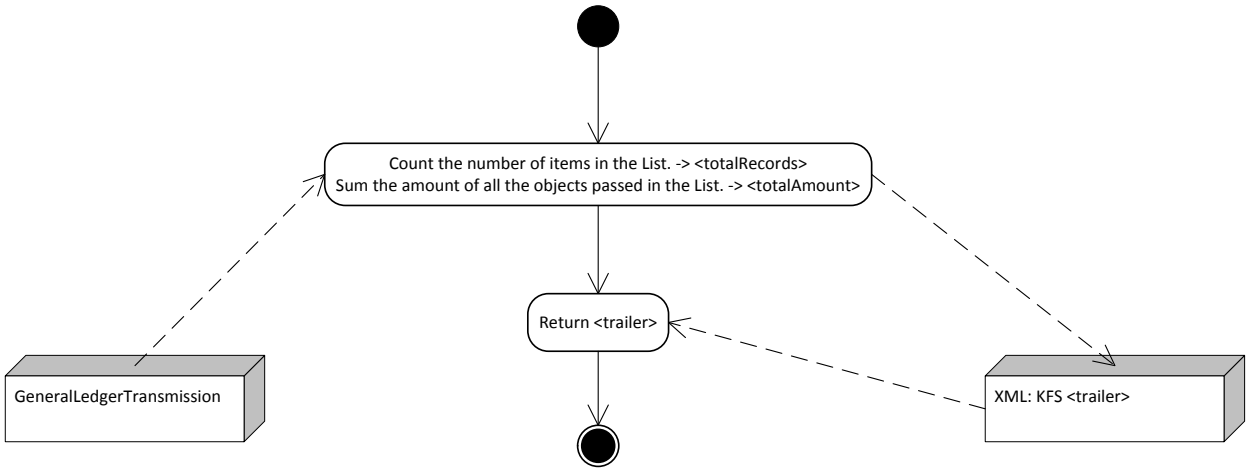
generateGeneralLedgerTransmissionHeader(batch)



generateGeneralLedgerTransmissionTransaction (generalLedgerTransmission, batch)



generateGeneralLedgerTransmissionTrailer (List <GeneralLedgerTransaction>, batch)



Information Service [InformationService]

Information service is responsible for Flags, Memos and Alerts.

Memos

getMemo (Long id)

Returns Memo.

Returns the Memo referenced by ID.

getMemos ()

getMemos (Long transactionId)

getMemos (String userId)

Returns List <Memo>

Returns the Memos, filtered by parameter.

getDefaultMemoLevel()

Returns Integer.

createNewMemo (String userId, String memo, Integer accessLevel, Date effectiveDate, Date expirationDate, Long previousMemoId)

createNewMemo (Long transactionId, String memo, Integer accessLevel, Date effectiveDate, Date expirationDate)

Returns Memo.

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. If effectiveDate is null, use current date.

If previousMemoId is passed, point created memo to previous memo, and point previous memo to new memo.

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

getFlags()

getFlags (String userId)

Returns List <Flag>

Return a list of flags/ flags by user.

createNewFlag (Long transactionId, Long flagTypeId, Integer accessLevel, Integer severity, Date effectiveDate, Date expirationDate)

createNewFlag (Integer transactionId, Long flagTypeId, Integer accessLevel, Integer severity, Date effectiveDate, Date expirationDate)

Returns Flag.

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

changeFlagSeverity (Flag flag, Integer newSeverity)

Returns Flag.

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 (Flag flag)

Returns Boolean.

If the flag is currently active, return true, otherwise return false.

Flag Types

persistFlagType (FlagType flagType)

Returns Integer id.

Creates a new entity if it didn't already exist (id is null) otherwise updates the old flagtype.

getFlagType (String code, String name, String namePattern)

Returns List<FlagType>.

Finds all FlagType that match all the passed parameters. Parameters can be null.

Alerts

getAlerts ()

getAlerts (String userId)

Returns List <Alert>.

Returns all alerts/ alerts for a user.

createAlert(Long transactionId, String alertText, Integer accessLevel, Date effectiveDate, Date expirationDate)

createAlert(String userId, String alertText, Integer accessLevel, Date effectiveDate, Date expirationDate)

Returns Alert.

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 alert, String newAlertText)

Returns Alert.

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

General

isEffective (Information information)

Returns Boolean.

Checks to see if the information is after the effectiveDate and before the expiredDate. If so, return TRUE.

changeLevel (Memo memo, Integer newLevel)

Returns Memo.

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()

Returns void.

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 information, Transcation transaction)

Returns Information.

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 information, Date newEffectiveDate)

setNewExpirationDate (Information information, Date newExpirationDate)

Returns Information.

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

getInformation (Long id)

Returns Information.

Returns the information referenced by ID.

getInformations()

getInformations(String userId)

Returns List <Information>

Return a list of Information objects, filtered by userId if passed.

persistInformation (Information information)

Returns Long id of information.

Persists the information in the database.

deleteInformation (long id)

Returns Boolean.

Delete the information in the database referenced by ID. If not found, return false.

Language Service [LanguageService]

persistLanguage(Language language)

Returns Integer id.

Saves or updates the passed language.

deleteLanguage (Long id)

Returns Boolean.

If the language exists, remove from database.

getLanguage (Long id)

Returns Language.

getLanguage (String locale)

Returns Language.

Returns language for the locale string.

getLanguages ()

Returns List <Language>

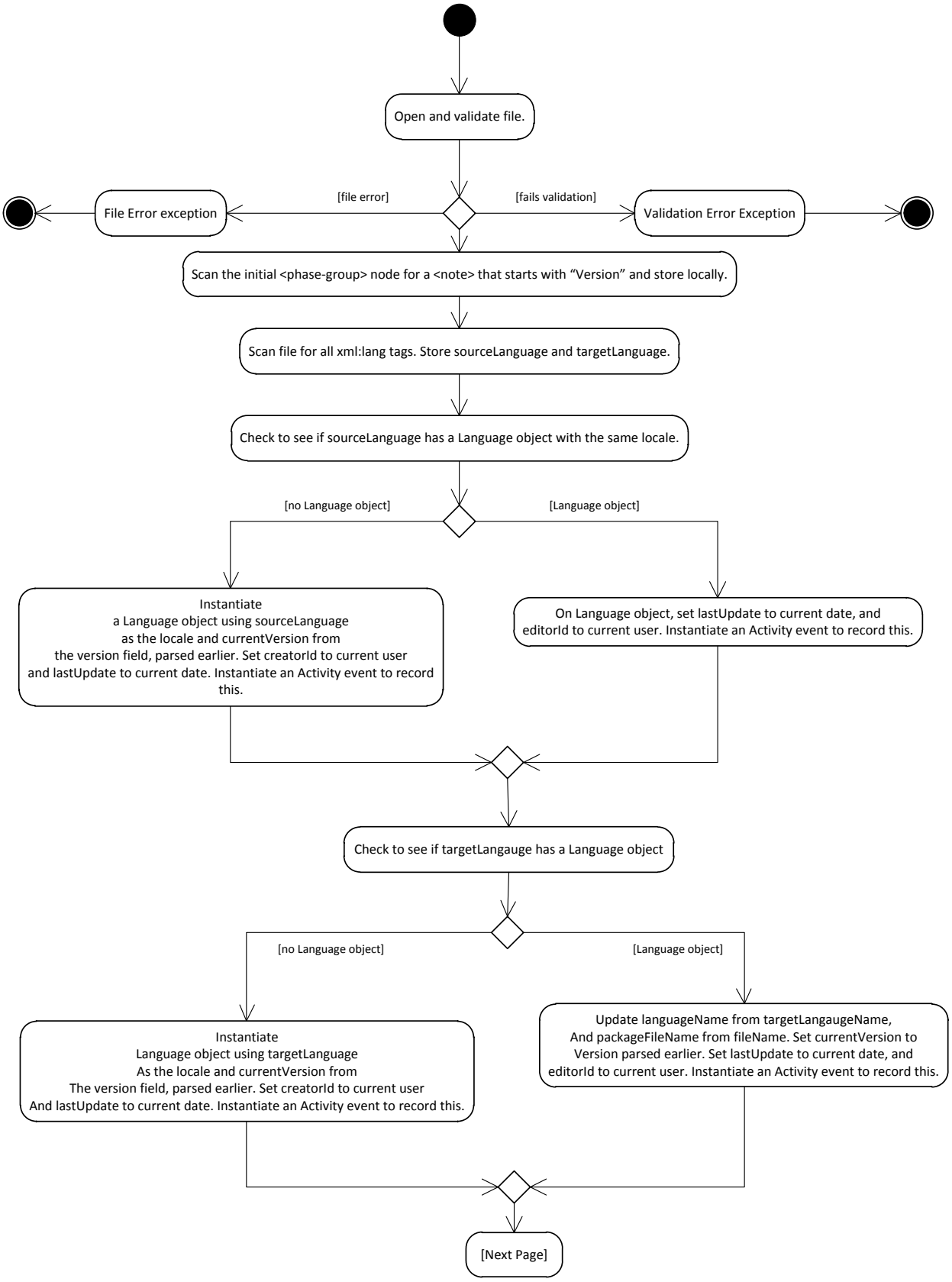
Localization Service

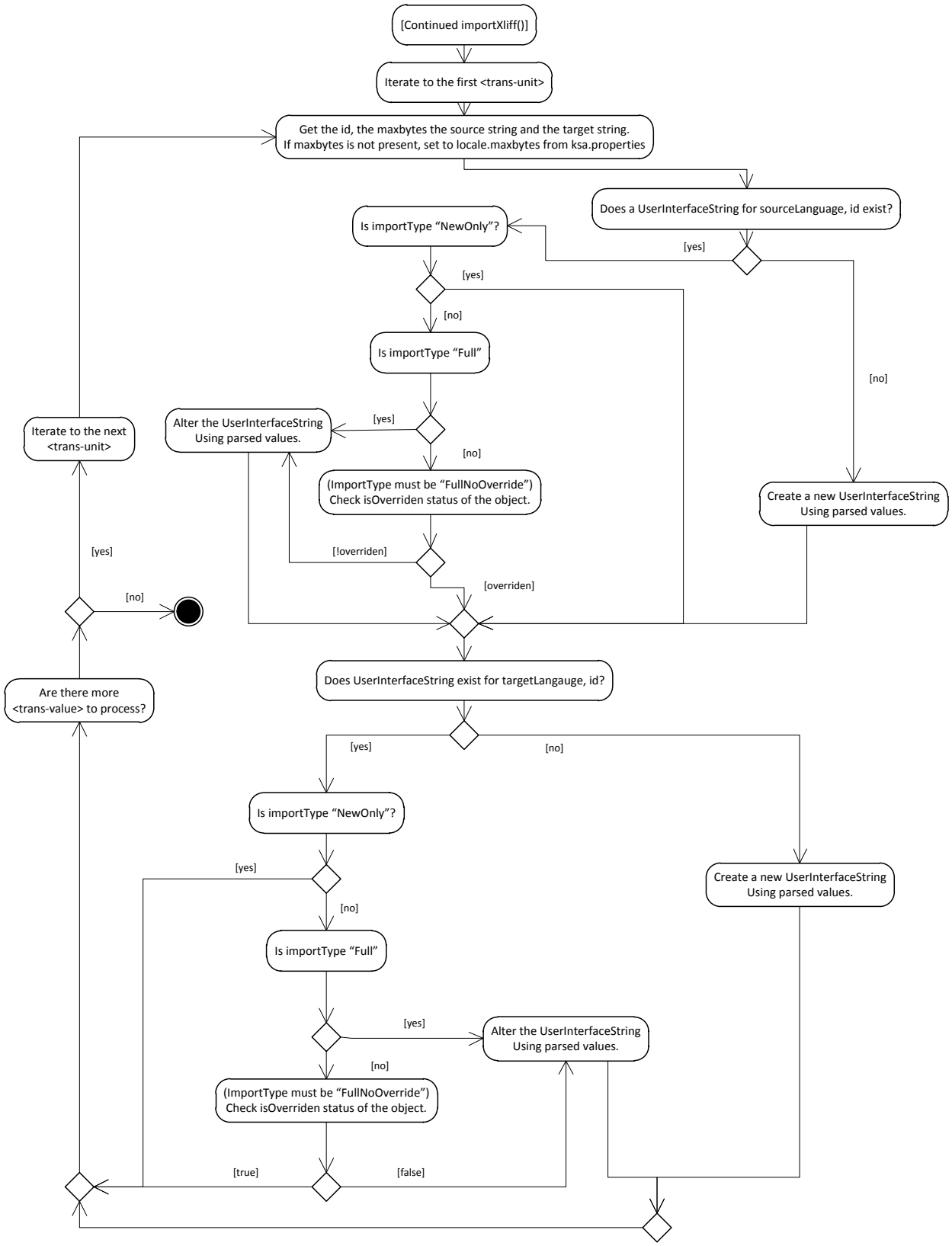
The localization service is used to import XLIFF files into the InstalledLanguage and UserInterfaceString objects to permit localization of the language of the user interface for KSA.

importResources (String content, importType [Full, FullNoOverride, NewOnly])

Returns List<LocalizedString>

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]
File Error exception
[fails validation]
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]
Alter the UserInterfaceString
Using parsed values.
[no]
(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]
Alter the UserInterfaceString
Using parsed values.
[no]
(ImportType must be “FullNoOverride”)
Check isOverriden status of the object.
[true]
[false]
Create a new UserInterfaceString
Using parsed values.

getLocalizedStrings (String locale)

Returns List <LocalizedString>

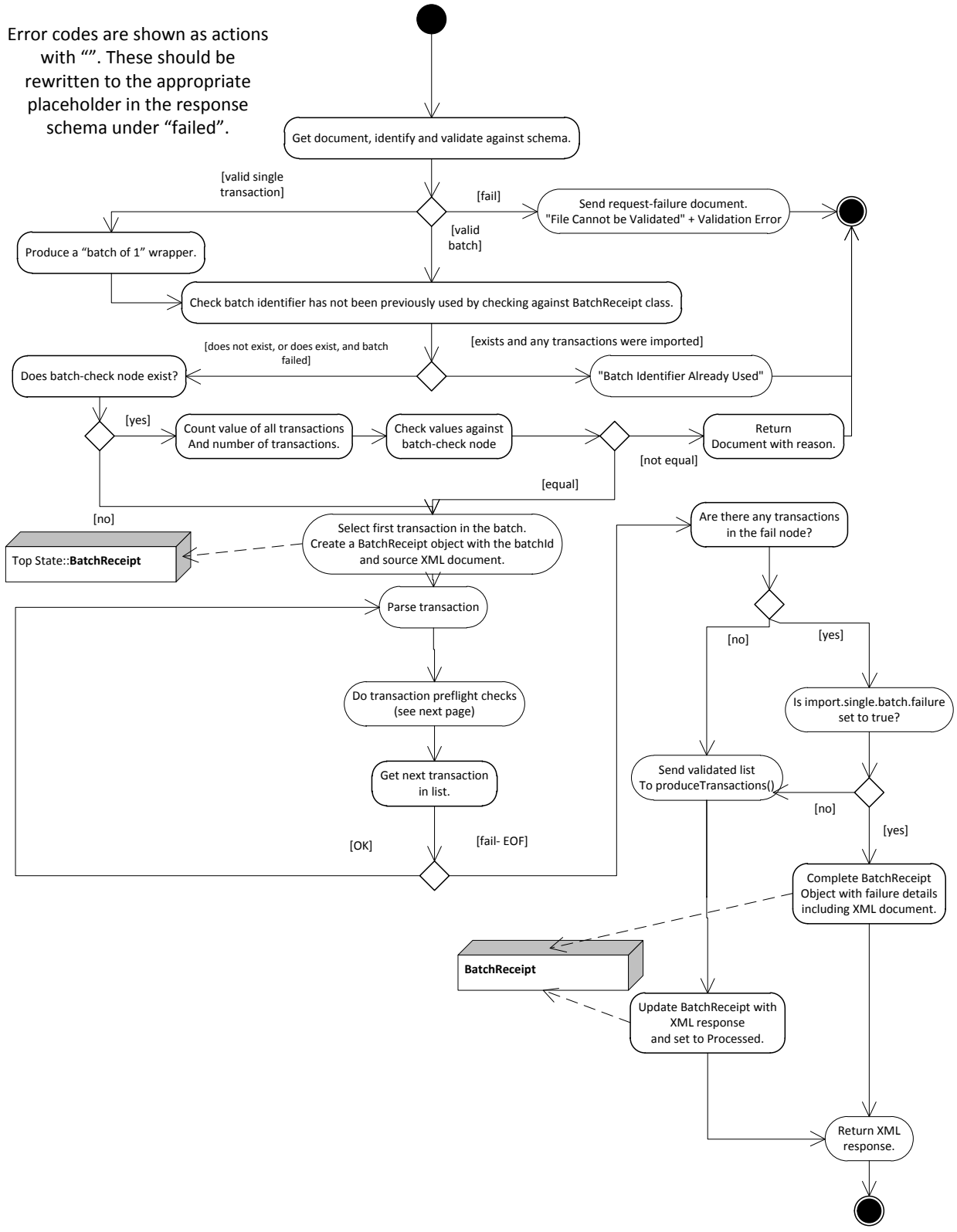
Returns all localized strings for the given locale.

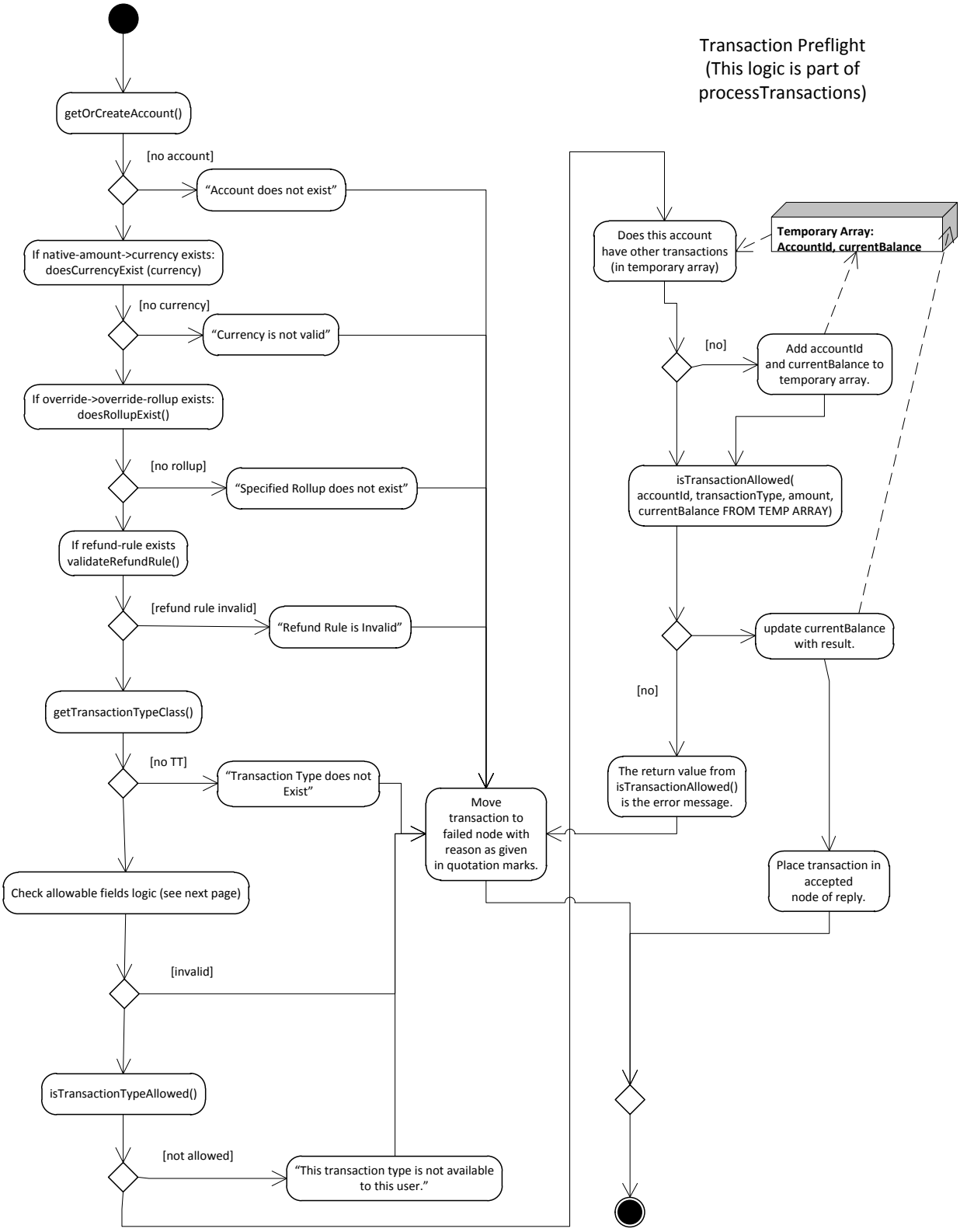
Transaction Import Service [TransactionImportService]

processTransactions (String xml)

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
[valid batch]
Produce a "batch of 1" wrapper.
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.
Parse transaction
Do transaction preflight checks (see next page)
Get next transaction in list.
[OK]
[fail- EOF]
Are there any transactions in the fail node?
[no]
[yes]
Is import.single.batch.failure set to true?
Send validated list To produceTransactions()
[no]
[yes]
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.

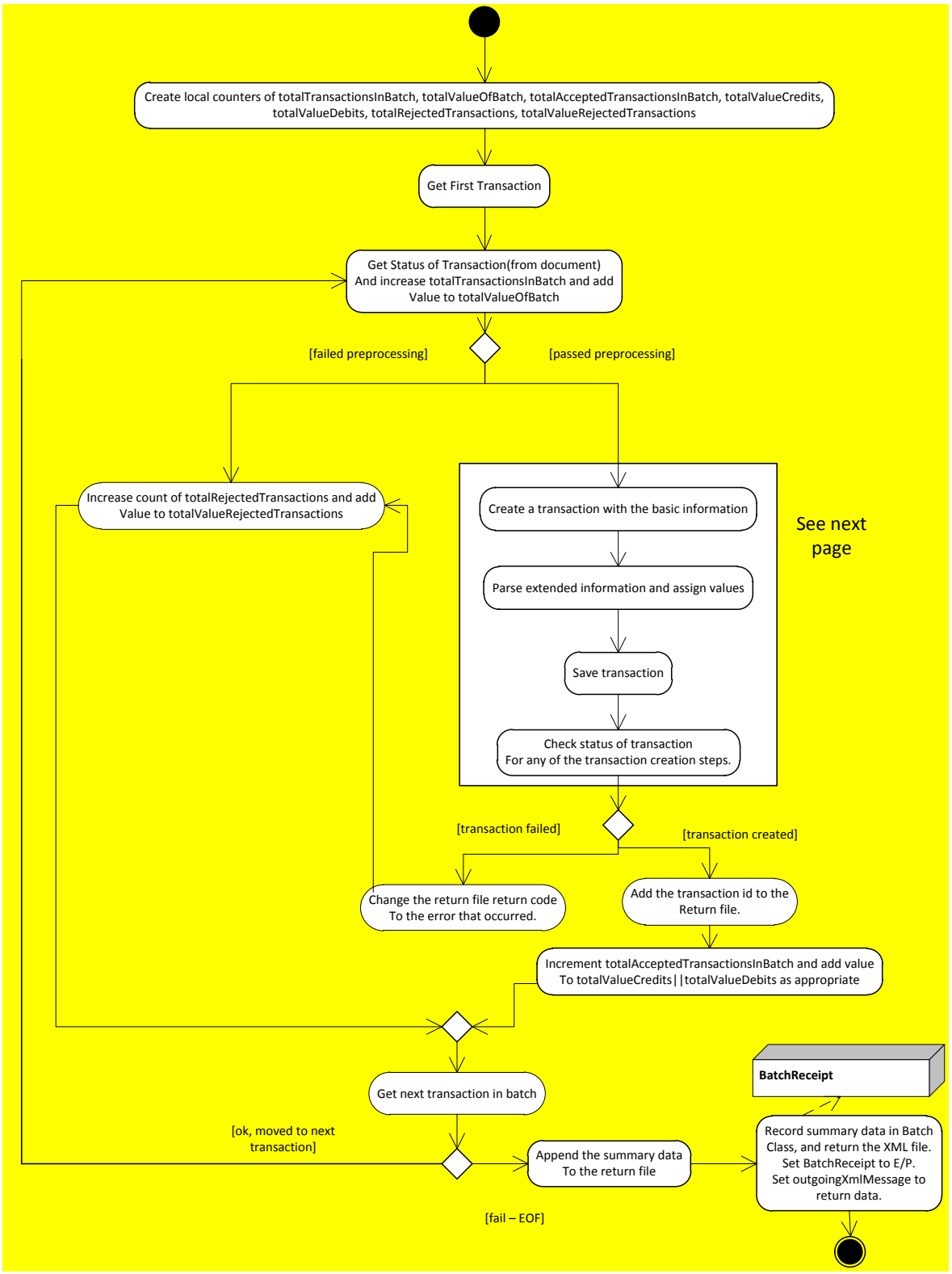
Allowable Fields Logic

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)

LOGIC MERGED.



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

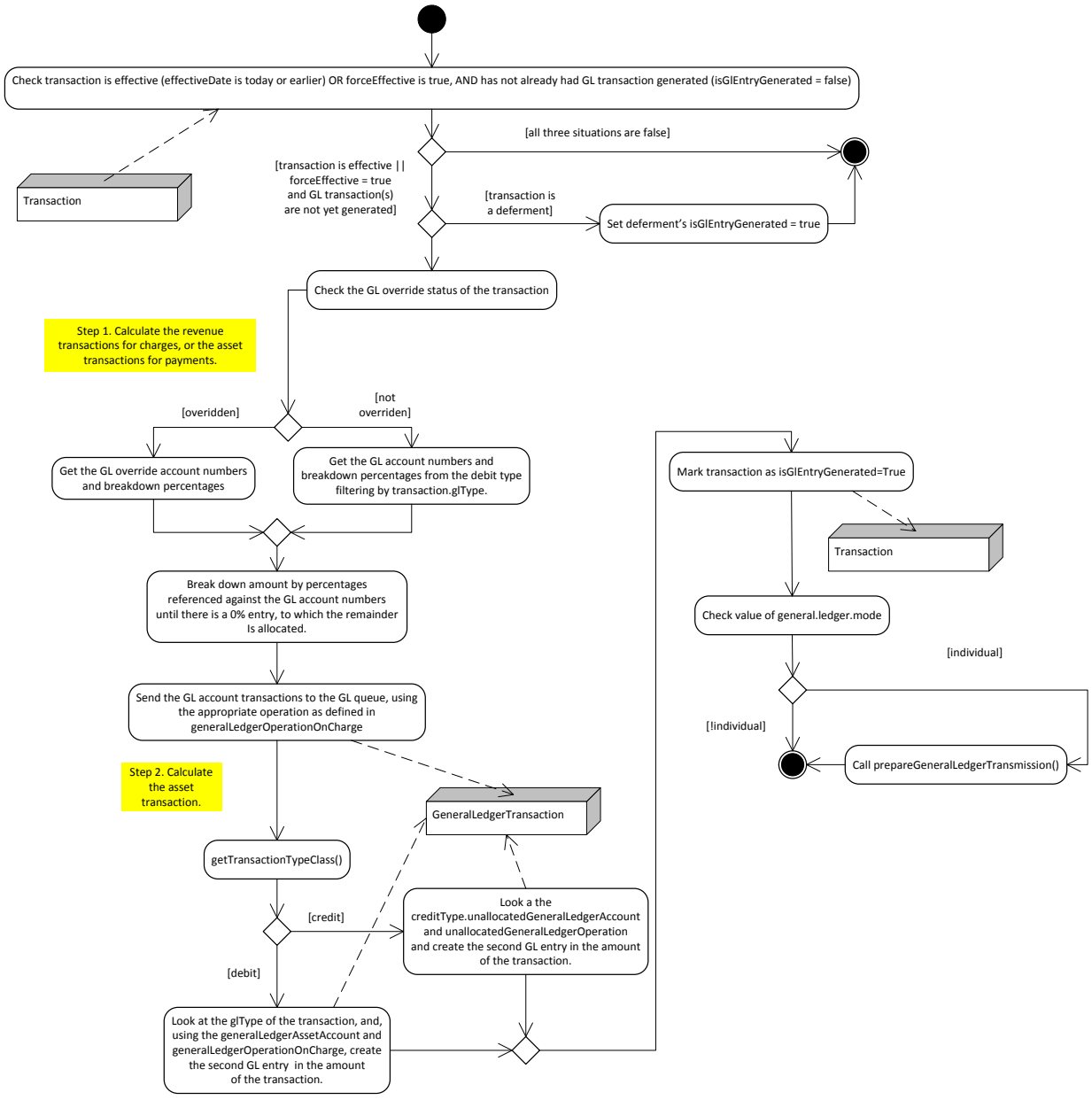
Default to general.ledger.type

Transaction Service

makeEffective (Long transactionId, Boolean forceEffective)

Returns void.

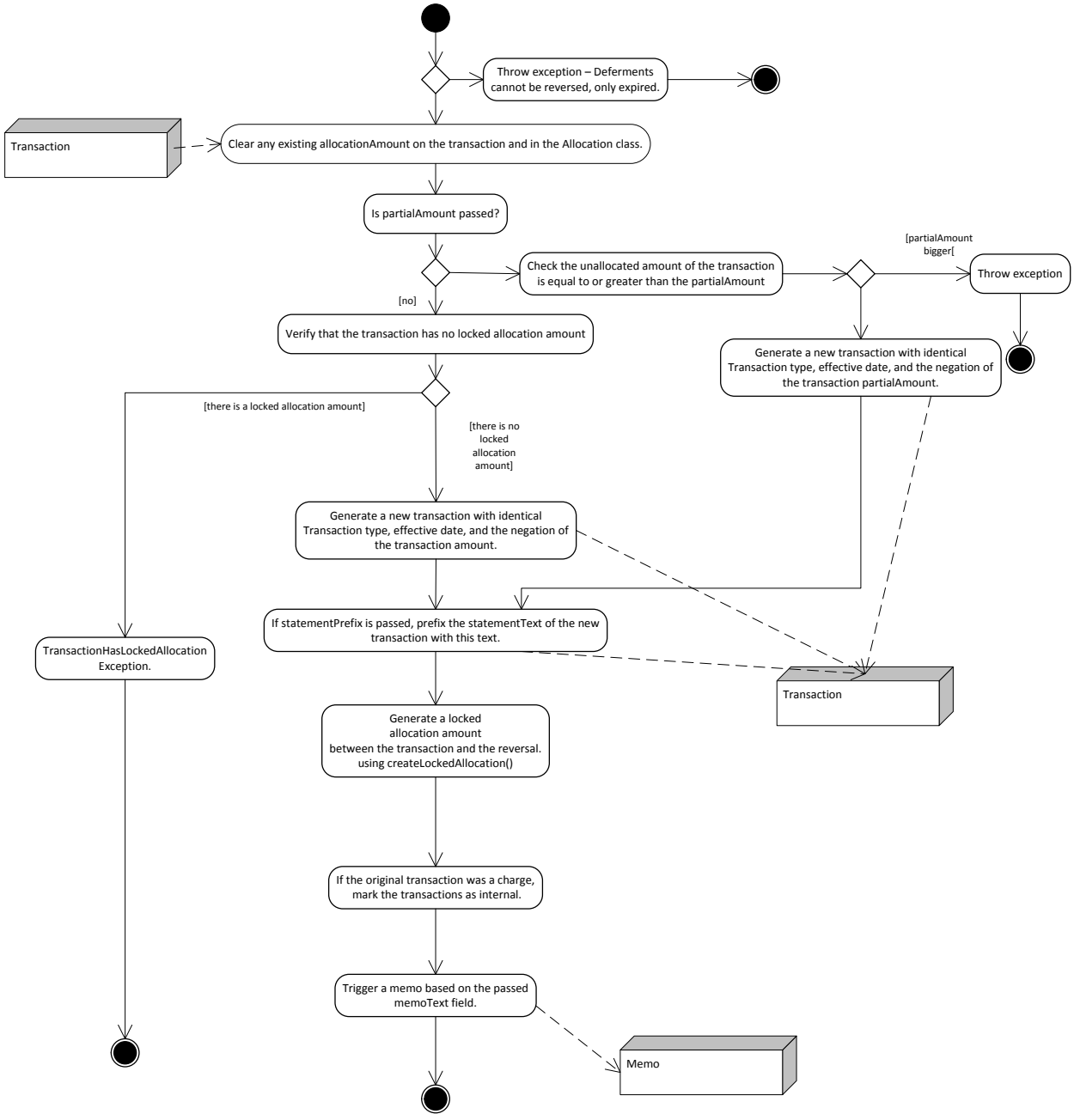
Moving a transaction from a pre-effective state to an effective state. Once a transaction is effective, its general ledger entries are created. In certain cases, a transaction might be moved to an effective state before its effective date, in which case, forceEffective is passed as true.



reverseTransaction(Long transactionId, String memoText, BigDecimal partialAmount, String statementPrefix)

Returns void.

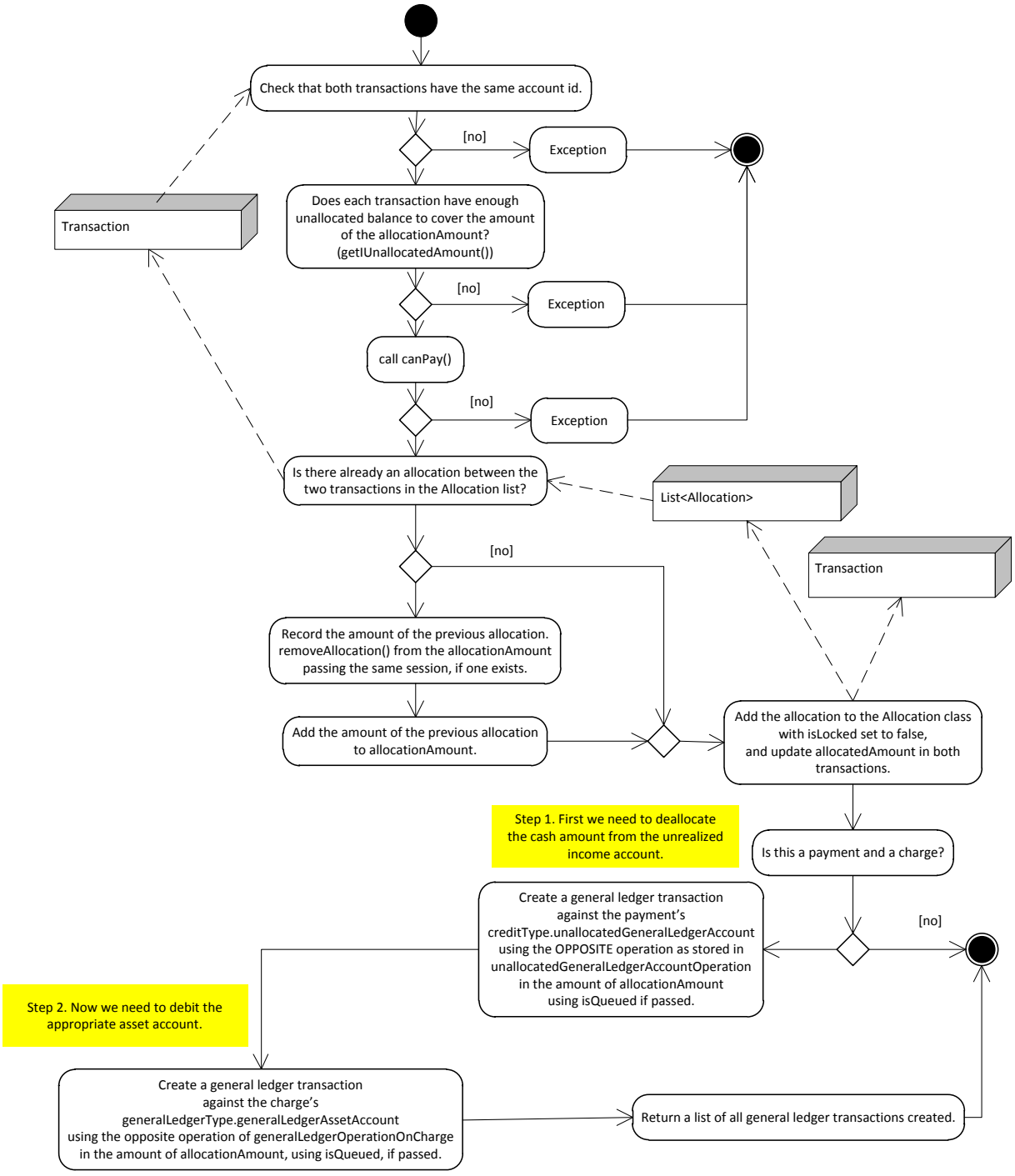
This is used to "reverse" a transaction, which it does by issuing an identical transaction with a negated amount. A memo is passed to the reverse operation, which is entered into the memo field.



createAllocation (Long transaction1, Long transaction2, BigDecimal amount)

createAllocation (Long transaction1, Long transaction2, BigDecimal amount, Boolean isQueued)

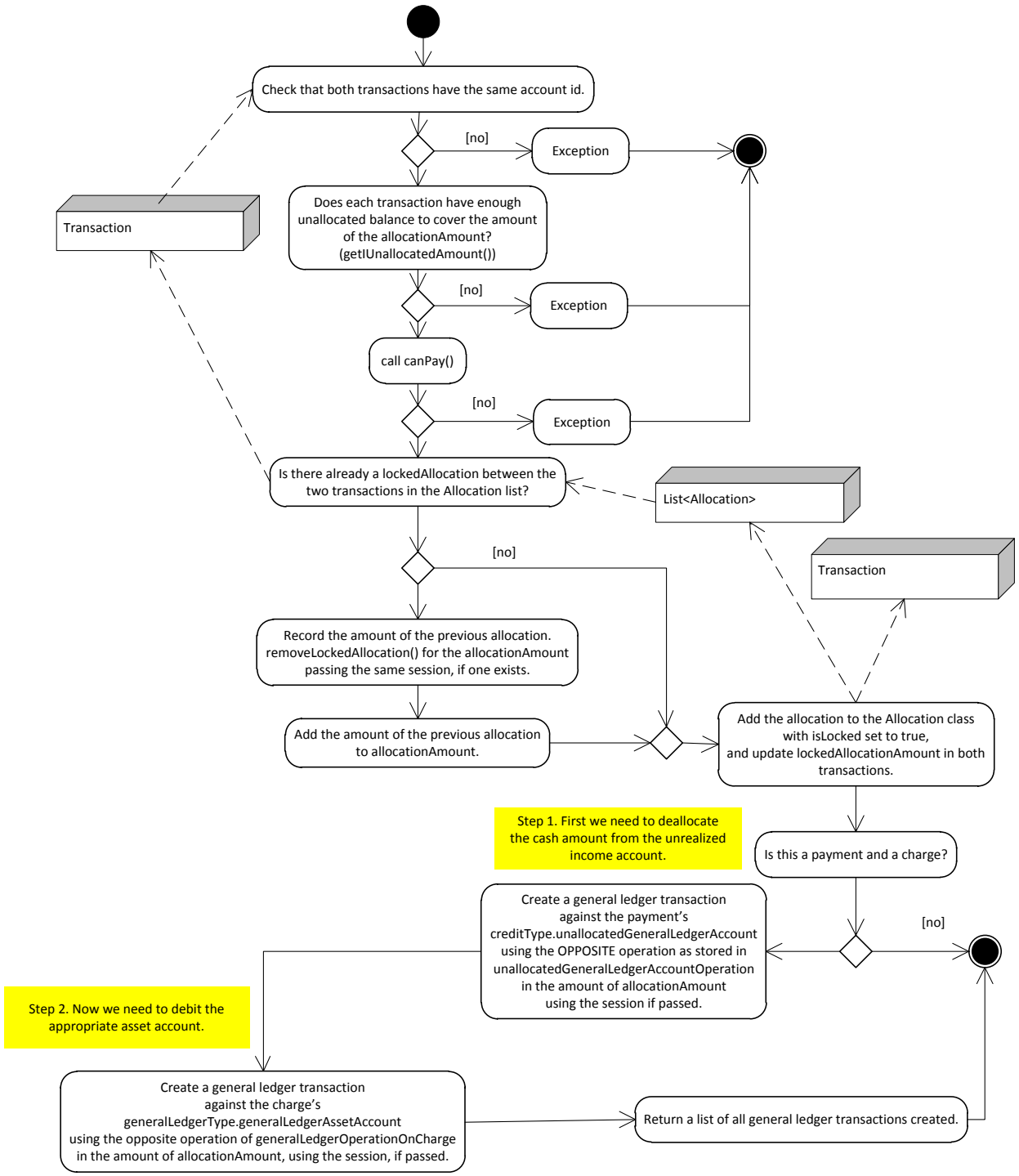
Returns CompositeAllocation.



createLockedAllocation (Long transaction1, Long transaction2, BigDecimal amount)

createLockedAllocation (Long transaction1, Long transaction2, BigDecimal amount, Boolean isQueued)

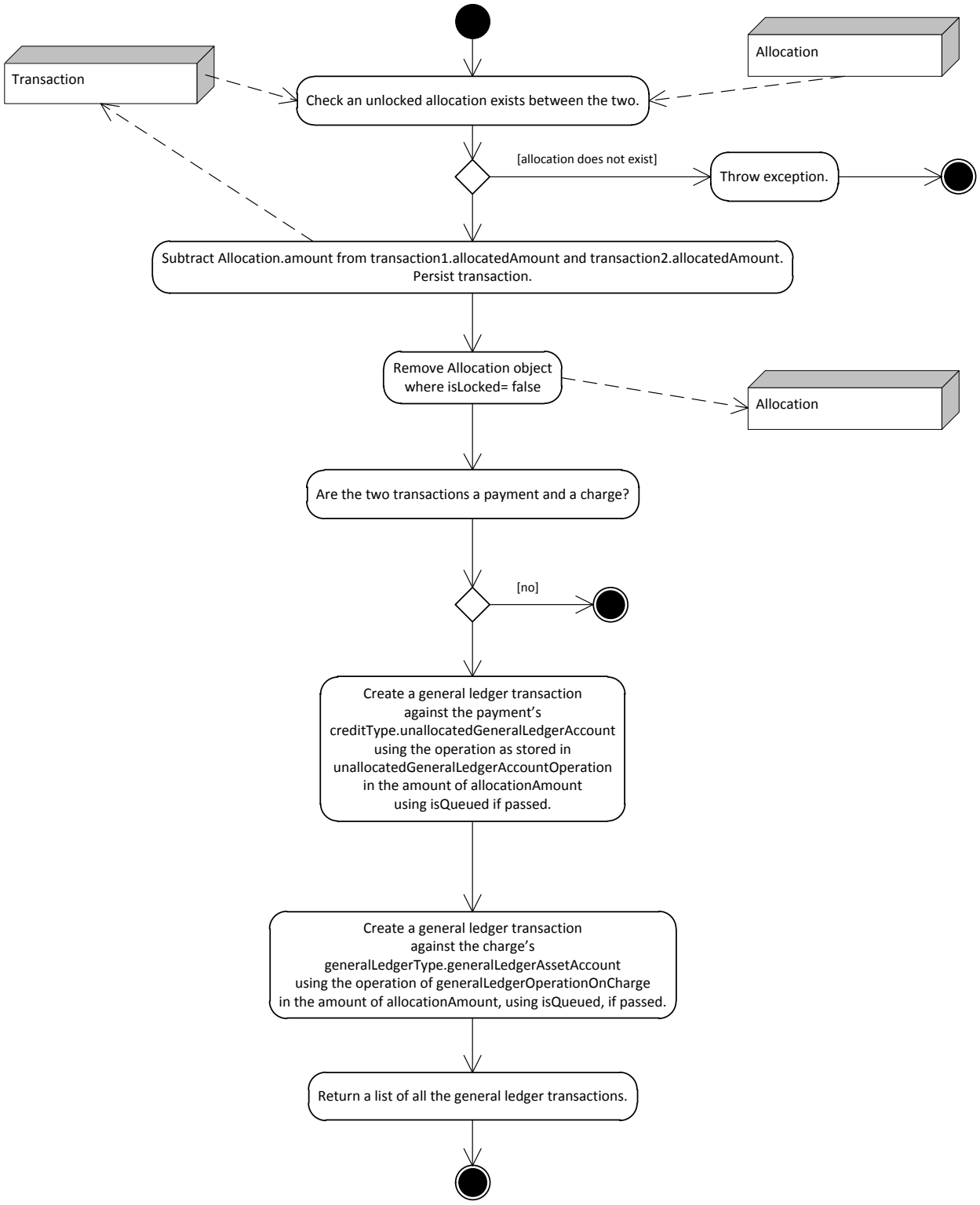
Returns CompositeAllocation.



removeAllocation (Long transactionId1, Long transactionId2)

removeAllocation (Long transactionId1, Long transactionId2, Boolean isQueued)

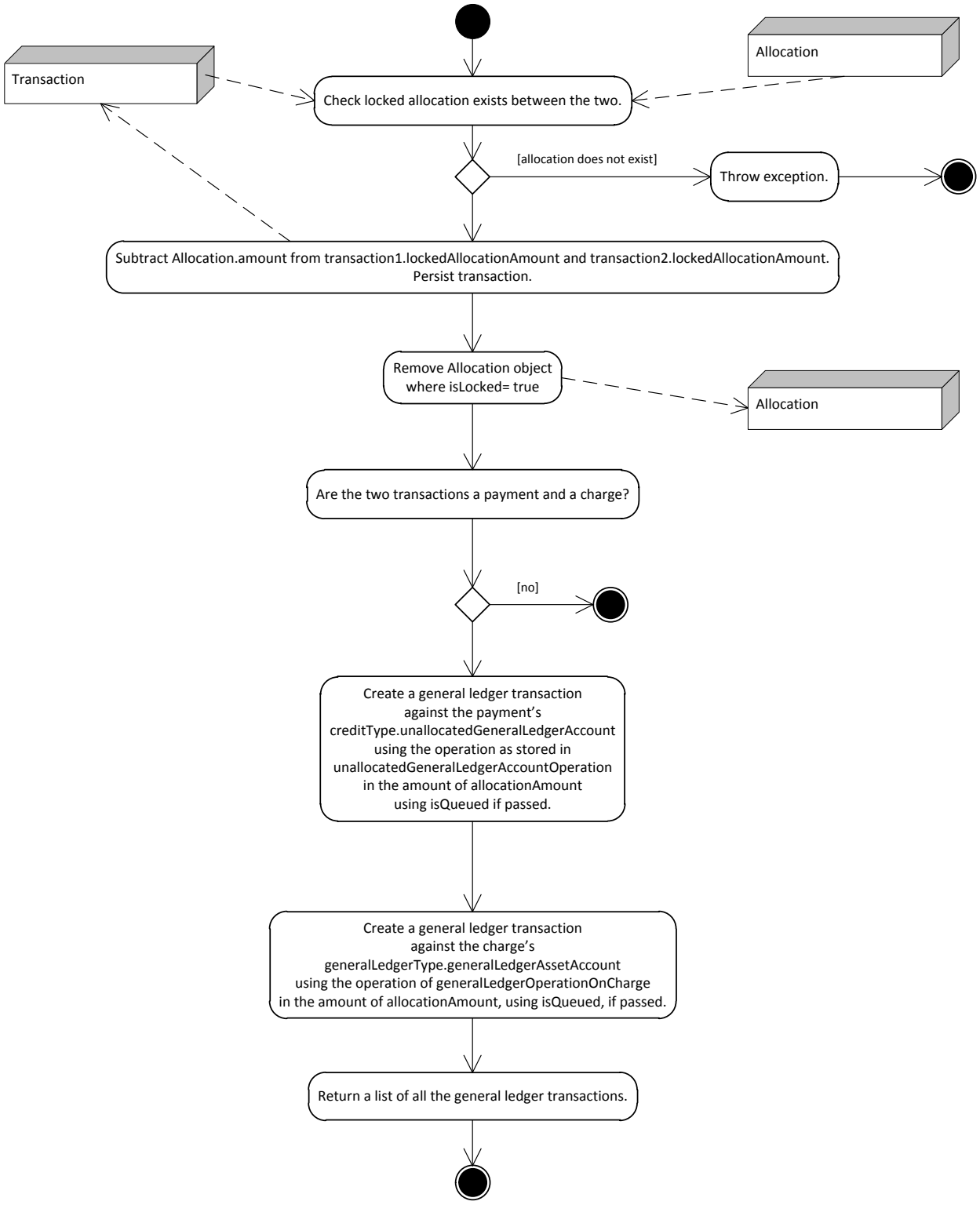
Returns List <GlTransaction>


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
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 (Long transactionId1, Long transactionId2)

removeLockedAllocation (Long transactionId1, Long transactionId2, Boolean isQueued)

Returns List <GlTransaction>



removeAllocations (Long transactionId)

removeAllAllocations (transactionId, isQueued)

Returns List <GlTransaction>

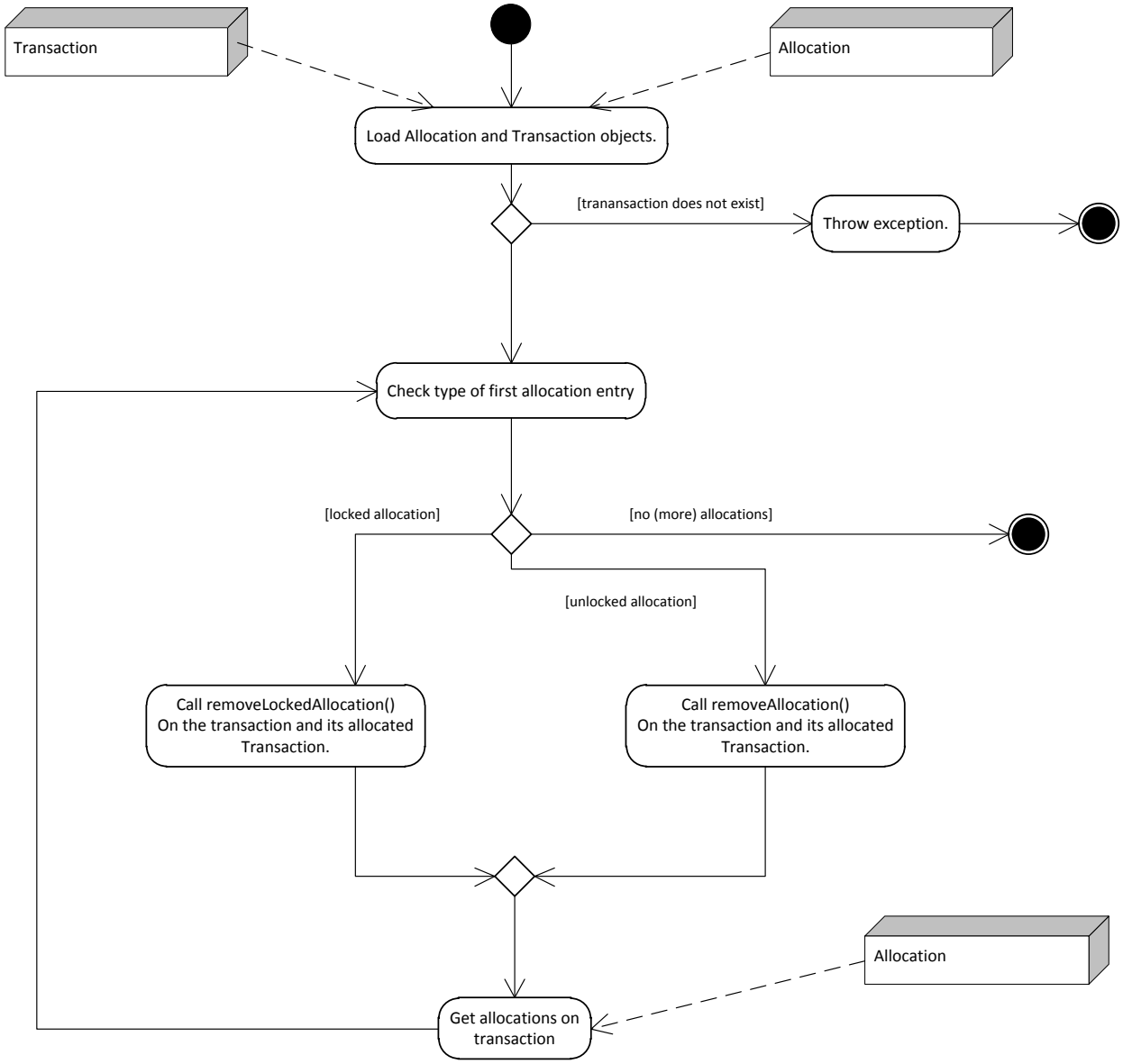
This method would most often be used when a payment has “bounced” and needs to be reversed off the account. A search is made of all allocations made against the transaction, and then those allocations are reversed. It is expected that the transaction referenced would then be reversed before a new payment 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 (Transaction transaction)

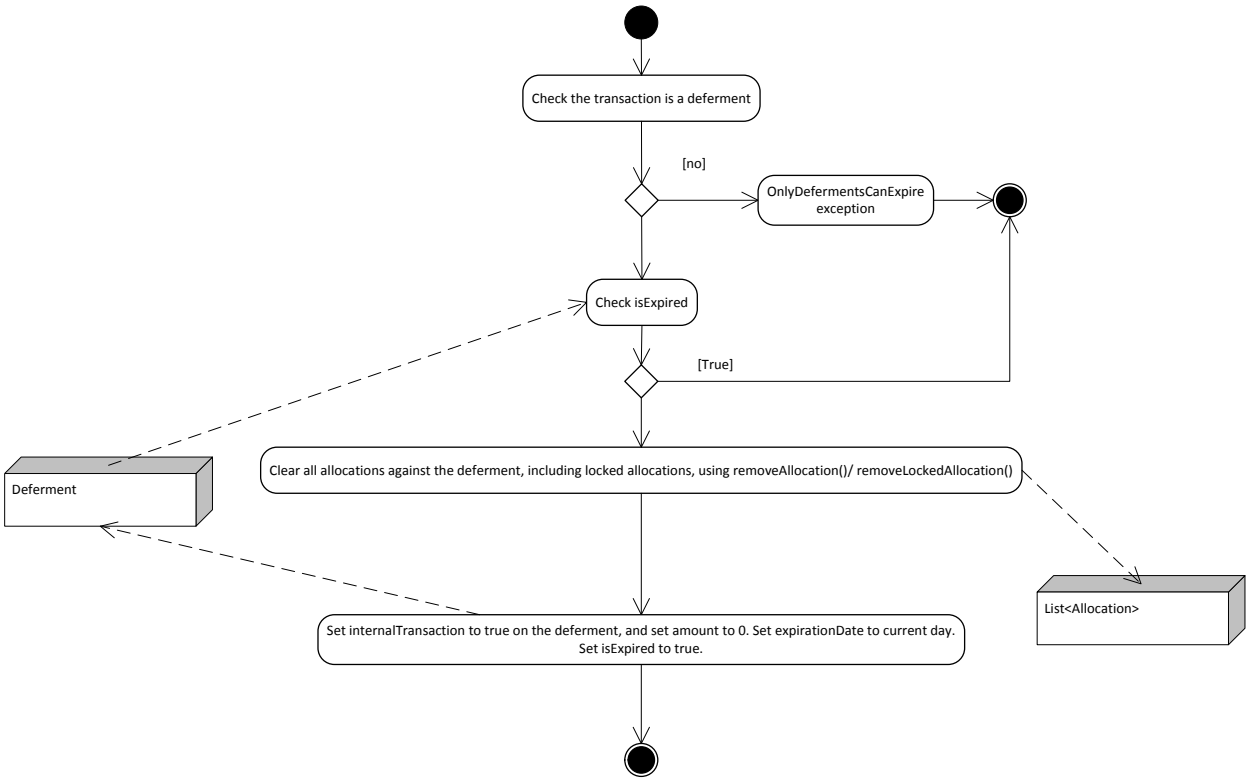
Returns BigDecimal.

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



expireDeferment(Long transactionId)

Returns void.



canPay (Long transaction1, Long transaction2)

canPay (Long transaction1, Long transaction2, int priority)

canPay(Long transaction1, Long transaction2, int priorityFrom, int priorityTo)

Returns Boolean.

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

Check the transactions are compatible:

One of the transactions is…	And the other transaction is…
Credit with positive amount	Debit with positive amount
Credit with positive amount	Credit with negative amount (payment reversal)
Debit with positive amount	Debit with negative amount (charge reversal)

If not, return false.

If the two transactions are charges, then return true.

If the two transactions are payments / deferments then return true.

If one transaction is a payment or a deferment, and the other is a charge, 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 (String transactionTypeId, String account, Date effectiveDate, BigDecimal amount)

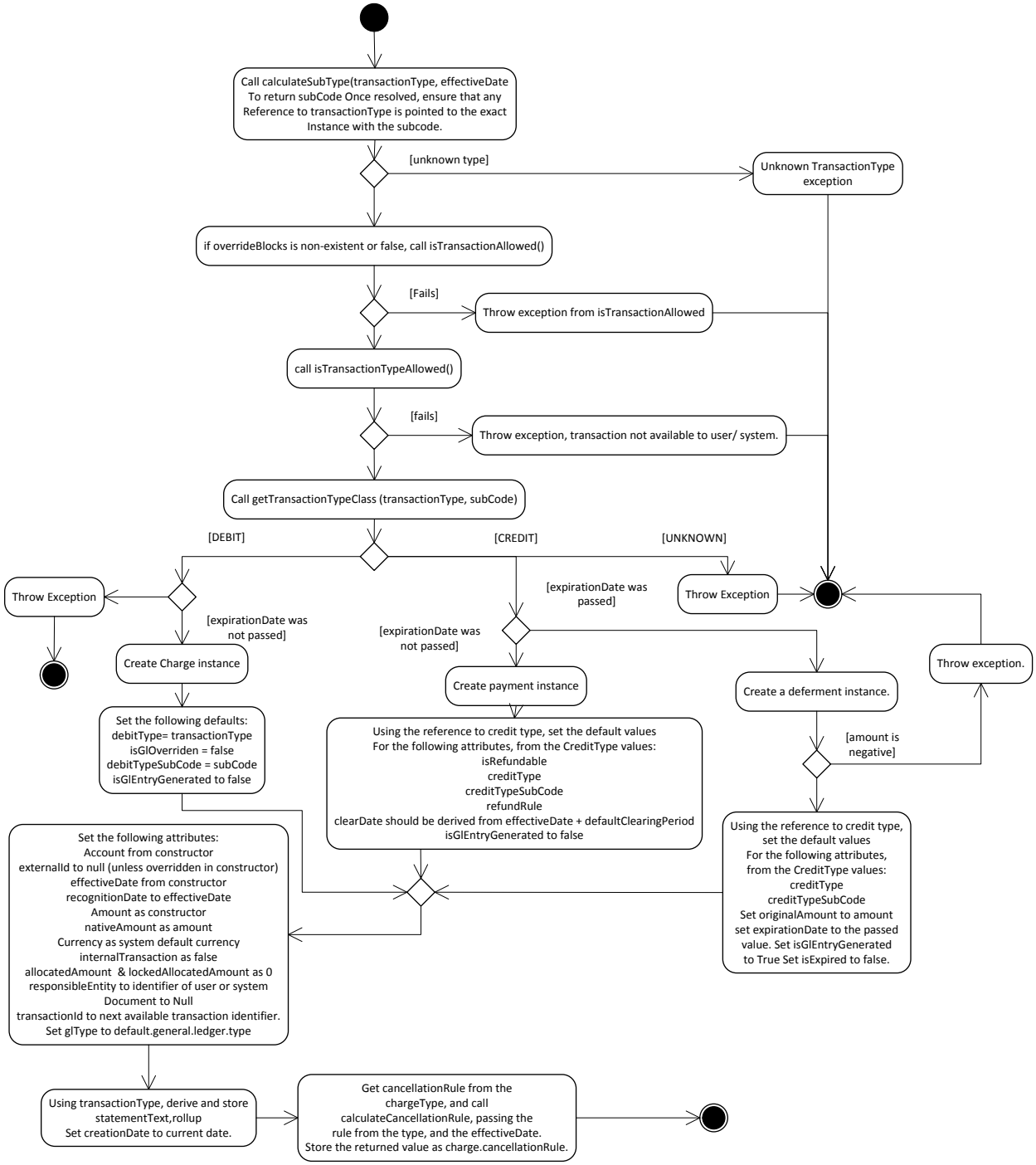
createTransaction (String transactionTypeId, String externalId, String account, Date effectiveDate, Date expirationDate, BigDecimal amount)

createTransaction (String transactionTypeId, String externalId, String account, Date effectiveDate, Date expirationDate, BigDecimal amount, Boolean overrideBlocks)

Returns Transaction.

Note that the transaction type will indicate if this is a payment or a charge. If a payment is passed with an expirationDate, then it will be treated as a deferment. If a charge is passed with an expiration date, then an exception will be thrown.

Note that deferments, unlike payments and charges, cannot be negative.



addTagToTransaction (transaction, tag)

Add the tag to the transaction's tag[] array.

removeTagFromTransaction (transaction, tag)

Ensure the tag is already part of the transaction otherwise throw exception.

Remove the tag from the Tag[] array of the transaction.

addRollupToTransaction (transaction, rollup)

Alter the transaction's rollup attribute to the passed rollup.

removeRollupFromTransaction (transaction)

Clear the rollup attribute of the passed transaction.

getTransactionType (Long transactionTypeId)

getTransactionType (Long transactionTypeId, effectiveDate)

Returns TransactionType.

Gets transaction type using the identifier.

getTransactionTypeClass (transactionTypeId)

Returns TransactionType

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

persistTransaction (Transaction transaction)

Returns Transaction.

Persist the transaction. If the transaction is a payment, with a tag of cash.tracking.tag, call the checkCashLimit() method.

getDaysBeforeDueDate()

Returns int.

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

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.

isTransactionAllowed(String accountId, String transactionTypeId, BigDecimal amount)

isTransactionAllowed (String accountId, String transactionType, BigDecimal amount, BigDecimal overrideCurrentBalance)

Returns Boolean.

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 (String accountId)

Returns void.

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

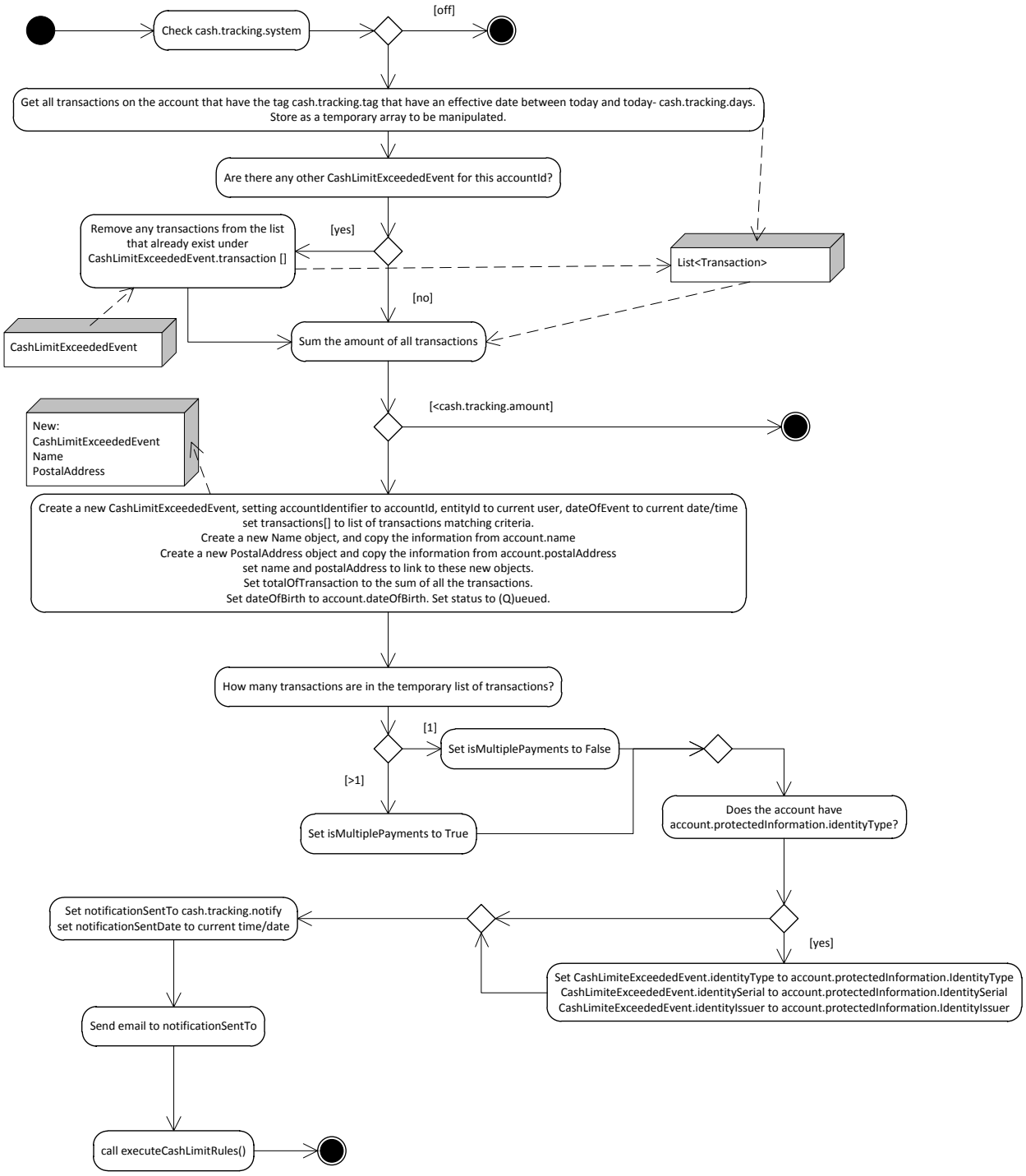
checkCashLimit (String accountId)

Returns Boolean.

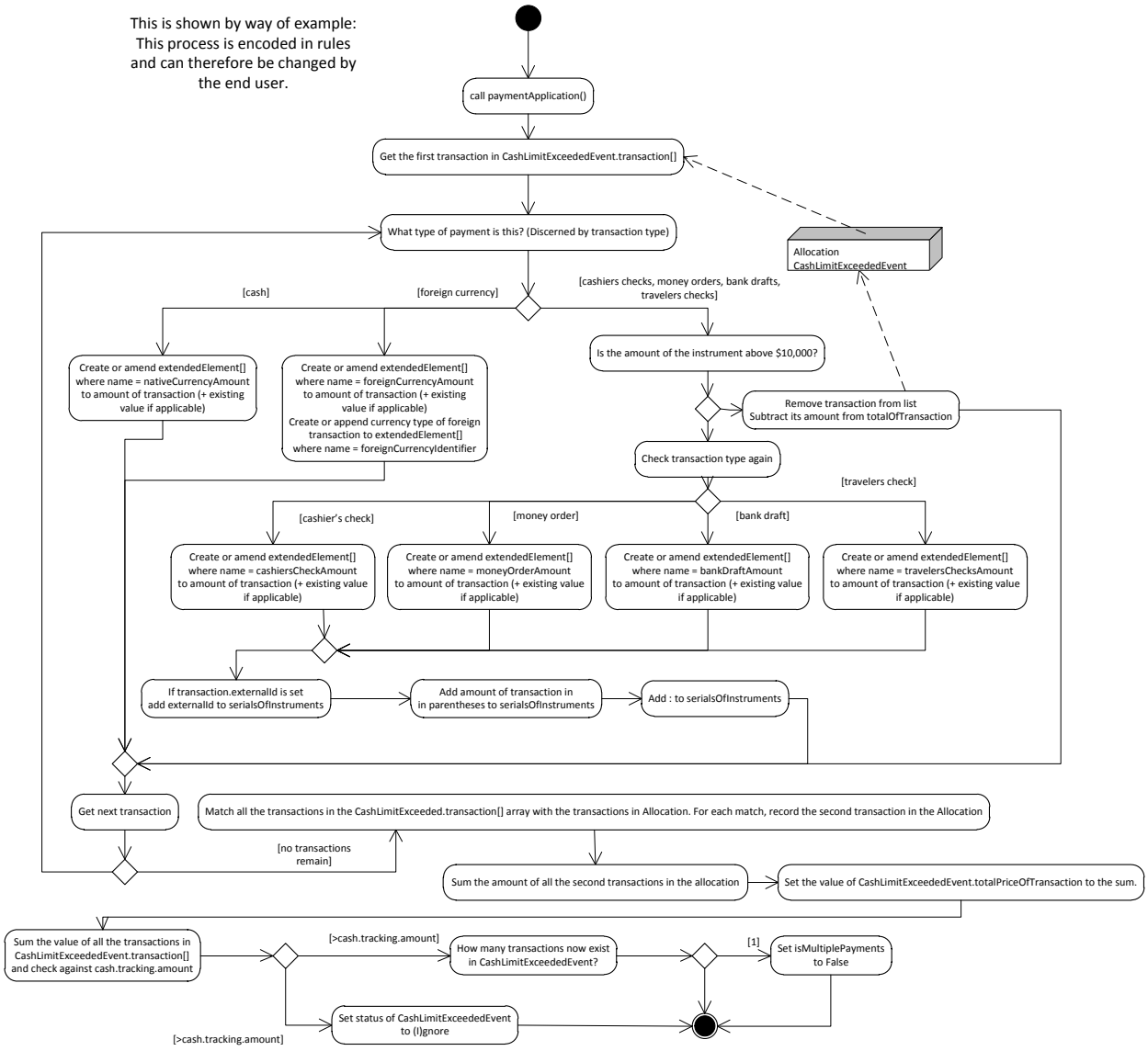
This process is triggered when a “cash” payment (tag is equal to cash.tracking.tag) is posted to an account. The system reviews the account for other cash payments, and if applicable, creates an event that is then passed to the cash limit rules.

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.

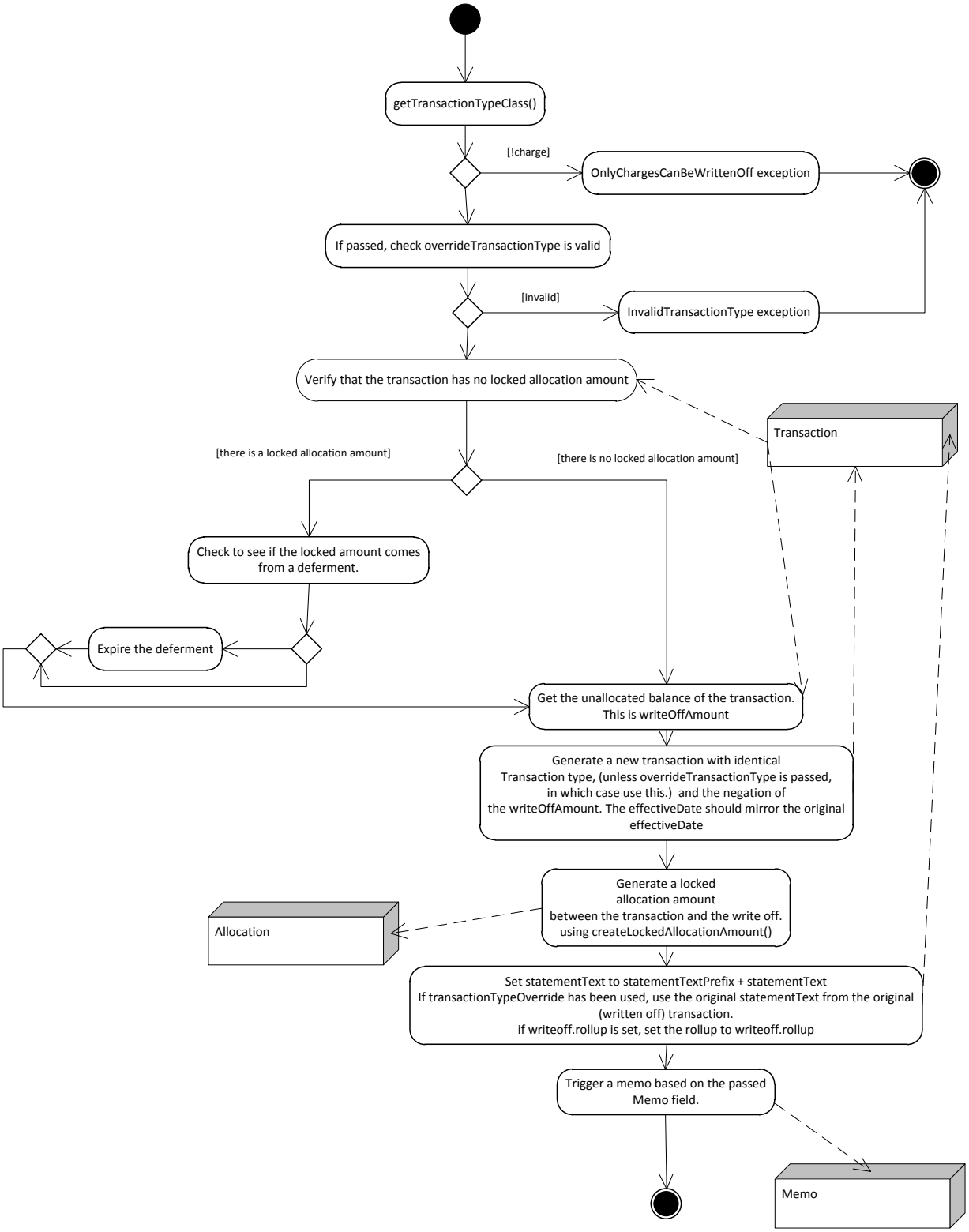

Check cash.tracking.system
[off]
Get all transactions on the account that have the tag cash.tracking.tag that have an effective date between today and today- cash.tracking.days.
Store as a temporary array to be manipulated.
Are there any other CashLimitExceededEvent for this accountId?
Remove any transactions 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?
[yes]
Set CashLimiteExceededEvent.identityType to account.protectedInformation.IdentityType
CashLimiteExceededEvent.identitySerial to account.protectedInformation.IdentitySerial
CashLimiteExceededEvent.identityIssuer to account.protectedInformation.IdentityIssuer
Set notificationSentTo cash.tracking.notify
set notificationSentDate to current time/date
Send email to notificationSentTo
call executeCashLimitRules()

executeCashLimitRules (cashLimitExceededEvent)



writeOffTransaction (Long transactionId, TranasctionTypeId transactionTypeId, String memoText, String statementPrefix)

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

cancelTransaction (Long transactionId, String memoText)

Returns void.

Check the trasnsactionId refers to a charge, otherwise throw exception.

Call getCancellationAmount() on the transaction.

Call reverseTransaction() passing the transaction and the result from getCancellationAmount().

Change charge.cancellationRule to “CANCELLED”

Create a memo using memoText. And point memo.transaction to the original transaction.

bounceTransaction (Long transactionId, String memoText)

Returns void.

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.

transactionExists (String userId, String transactionTypeId)

transactionExists (String userId, String transactionTypeId, Date dateFrom, Date dateTo)

transactionExists (String userId, String transactionTypeId, BigDecimal amountFrom, BigDecimal amountTo)

transactionExists (String userId, String transactionTypeId, Date dateFrom, Date dateTo, BigDecimal amountFrom, BigDecimal amountTo)

Returns Boolean.

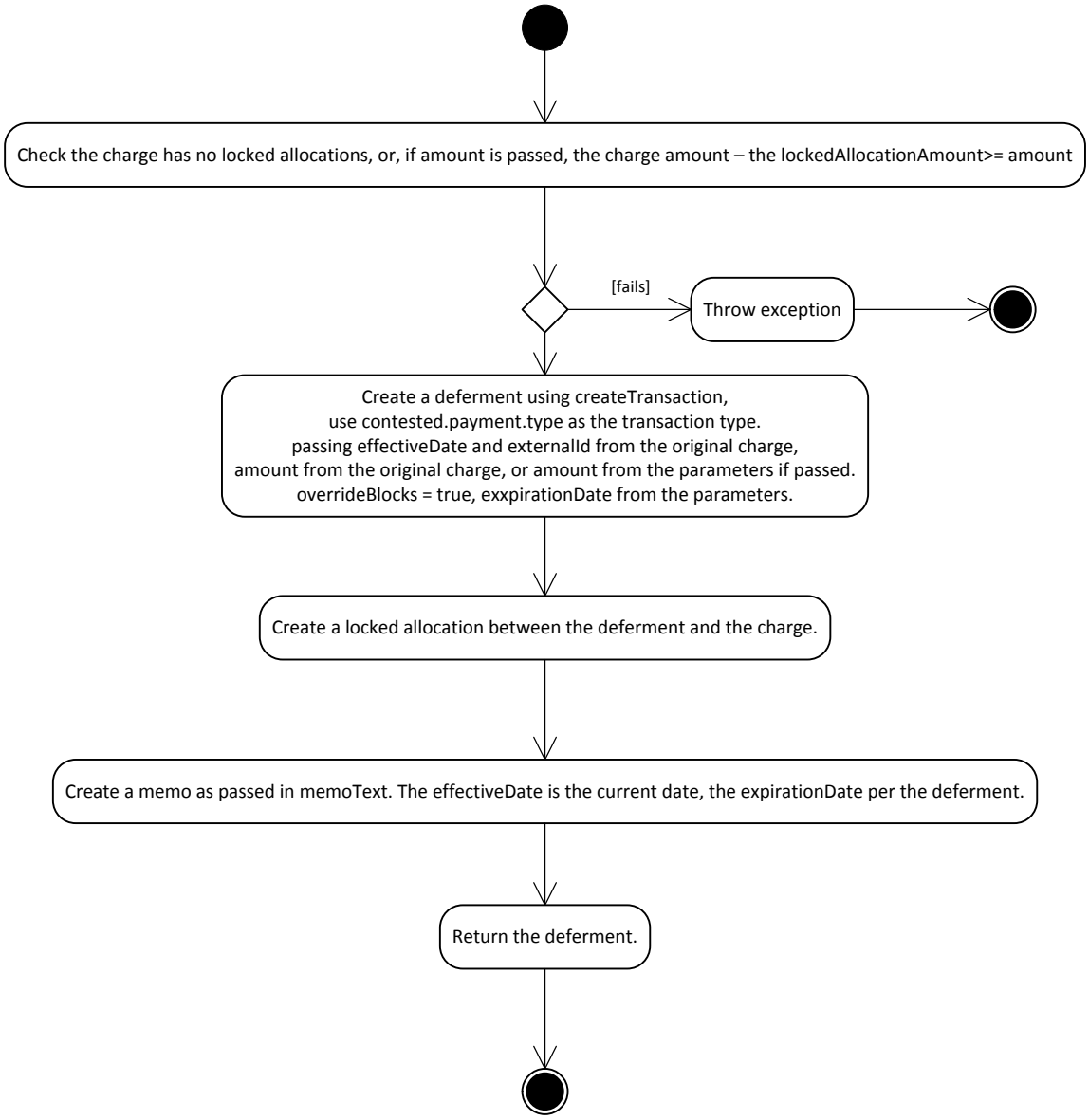
These methods are provided to assist the fee management service decided if it needs to reassess a certain fee.

Allows the fee management rules locate a transaction that already exists. This returns true if the transaction that meets the criteria of the method is found. Otherwise it returns false. This would be used for example, to check that a prepayment has been made before fees are assessed on a student.

Return true if the transaction indicated exists, otherwise return false.

contestCharge (Charge charge, Date expirationDate, String memoText)

Returns Deferment.



findTransactionsByStatementPattern (String pattern)

Returns List<Transaction>

Returns a list of transactions that match the pattern against the statementText.

getCharge(Long id)

Returns Charge.

Returns the charge with the passed identifier.

getCharges()

getCharges(String userId)

Returns List<Charge>

Returns a list of all charges/ charges on an account.

getPayment(Long id)

Returns Payment.

Returns the payment with the passed identifier.

getPayments()

getPayments(String userId)

Returns List<Payment>

Returns a list of all payments/payments on an account.

getDeferment(Long id)

Returns Deferment.

Returns the deferment with the passed identifier.

getDeferments()

getDeferments(String userId)

Returns List<Deferment>

Returns a list of all deferments/ deferments on an account.

getTransaction(Long id)

Returns Transaction.

Returns the Transaction with the passed identifier.

getTransactions()

getTransactions(String userId)

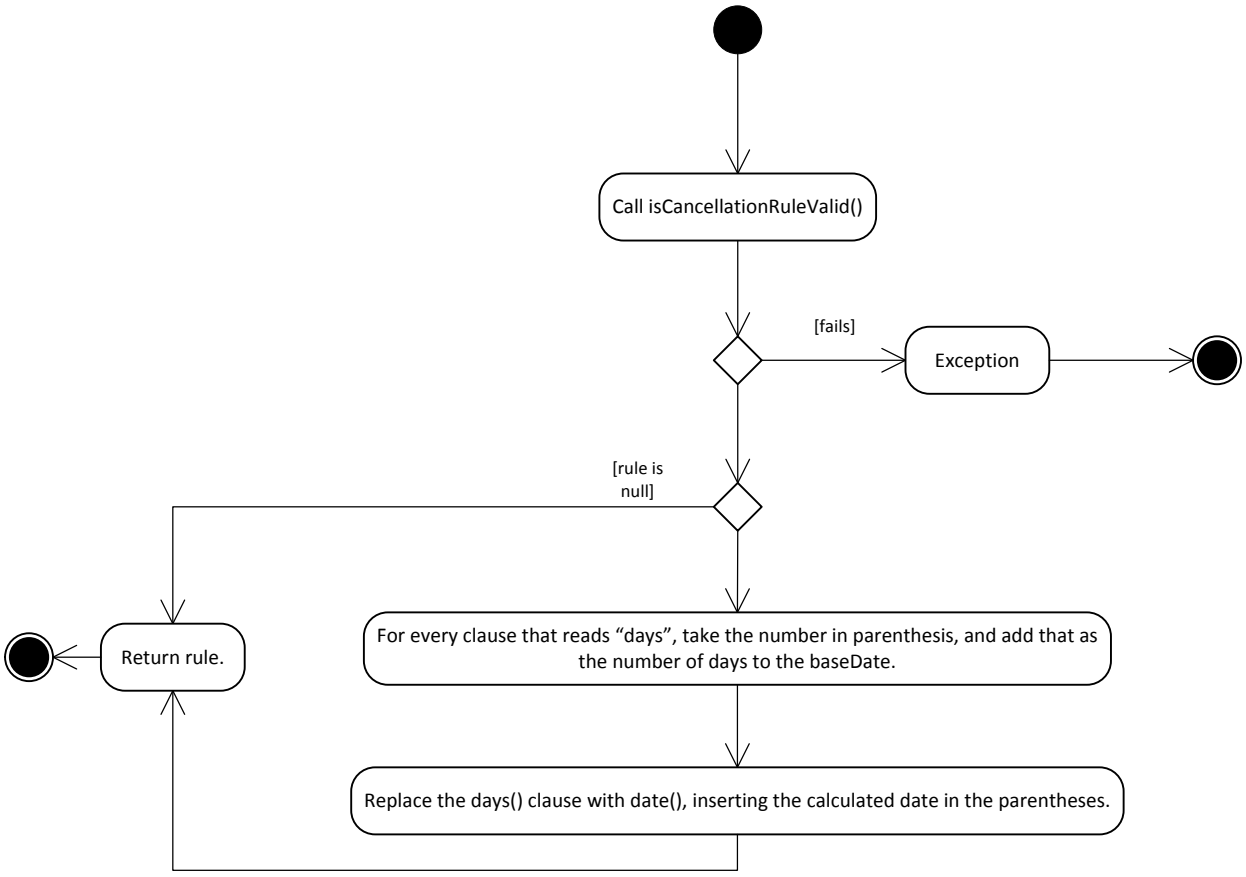
Returns List<Transaction>

Returns a list of all Transactions/ Transactions on an account.

calculateCancellationRule (String cancellationRule, Date baseDate)

Returns String.

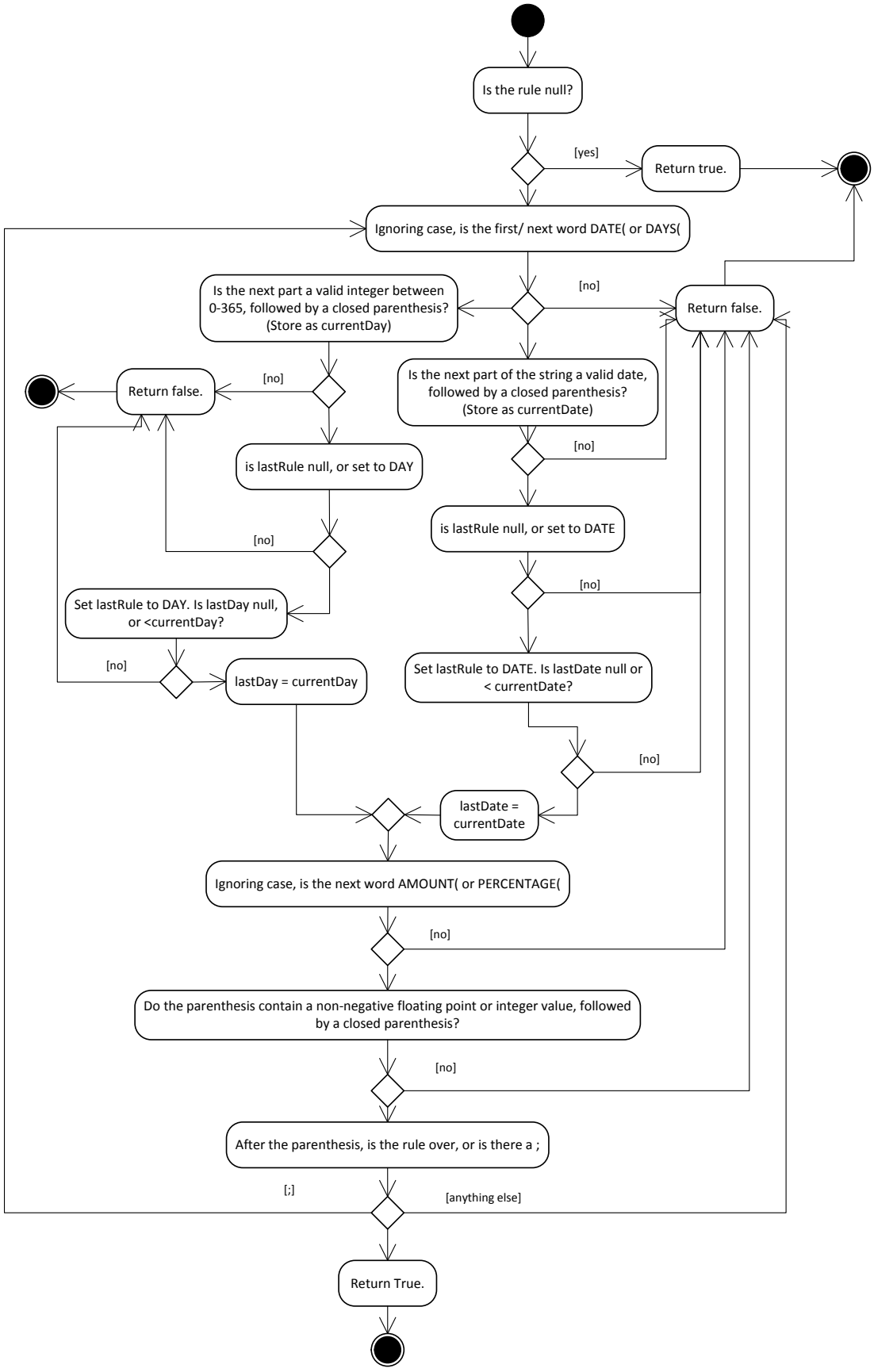
Takes the cancellationRule from the debit transaction type, and, using the baseDate, calculates the appropriate dates to be stored in the actual transaction version of the cancellationRule.



isCancellationRuleValid (String cancellationRule)

Returns Boolean.

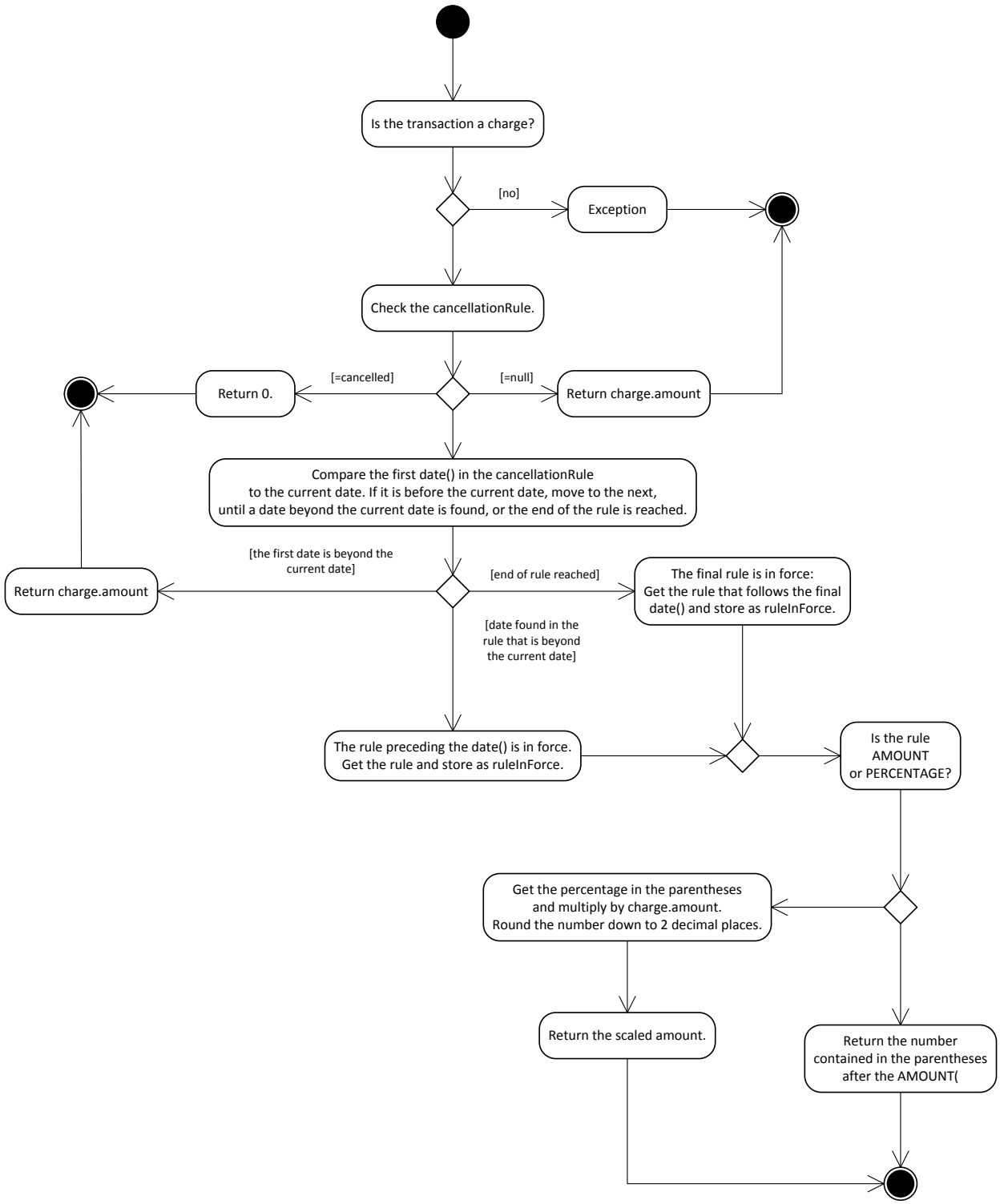
Checks to see if the rule being entered is legal.


Is the rule null?
[yes]
Return true.
Ignoring case, is the first/ next word DATE(or DAYS(
Is the next part a valid integer between 0-365, followed by a closed parenthesis? (Store as currentDay)
[no]
Return false.
Return false.
[no]
Is the next part of the string a valid date, followed by a closed parenthesis? (Store as currentDate)
[no]
is lastRule null, or set to DAY
is lastRule null, or set to DATE
[no]
[no]
Set lastRule to DAY. Is lastDay null, or <currentDay?
Set lastRule to DATE. Is lastDate null or < currentDate?
[no]
[no]
lastDay = currentDay
lastDate = currentDate
Ignoring case, is the next word AMOUNT(or PERCENTAGE(
[no]
Do the parenthesis contain a non-negative floating point or integer value, followed by a closed parenthesis?
[no]
After the parenthesis, is the rule over, or is there a ;
[;]
[anything else]
Return True.

getCancellationAmount (Charge charge)

Returns BigDecimal.

Using the cancellationRule, calculates the appropriate amount that can be cancelled from a charge.



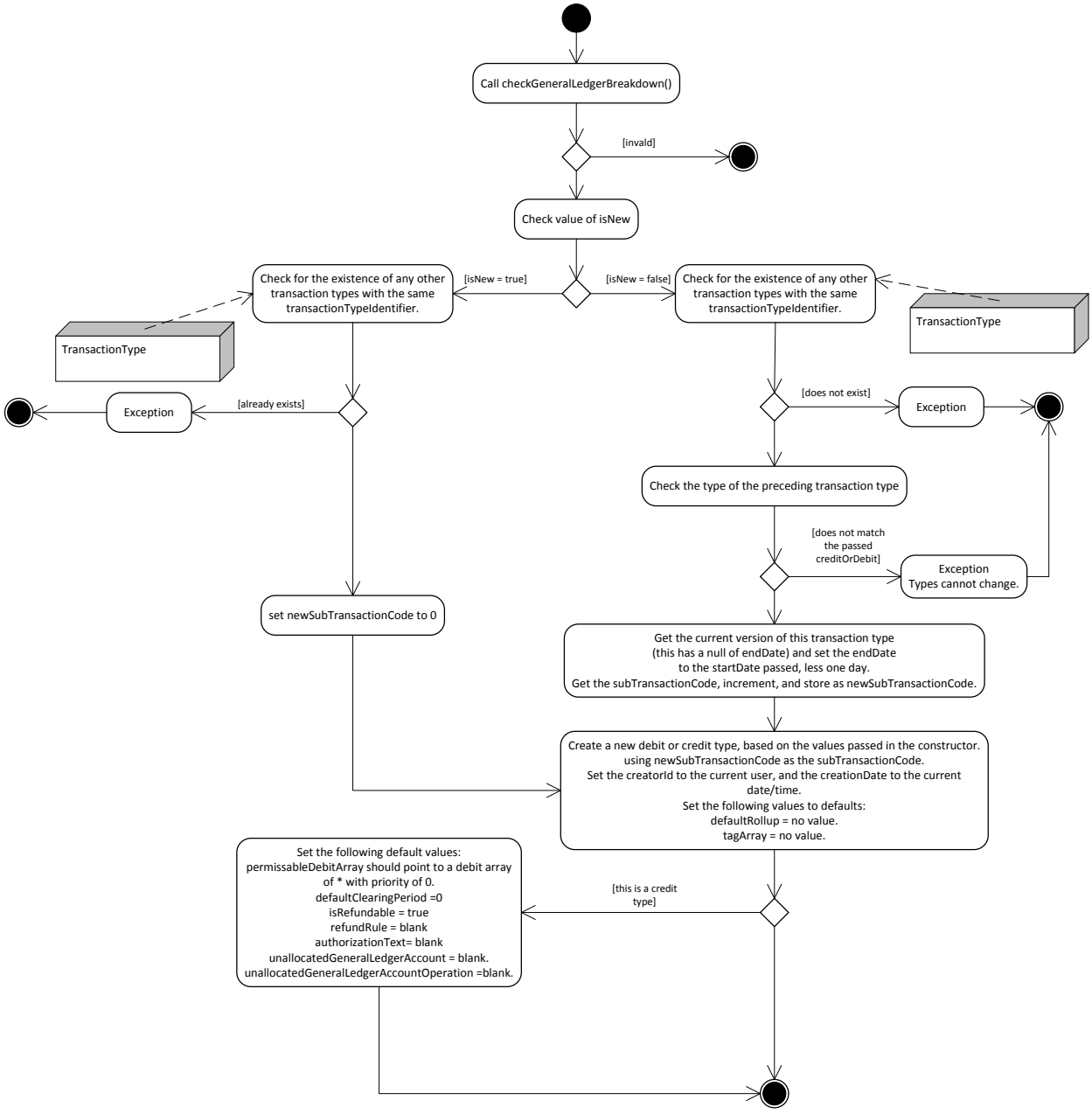
Transaction Type Service (TransactionService)

Methods relating to the creation of transaction types.

createTransactionType (Boolean isNew, GlOperation creditOrDebit, Long transactionTypeIdentifier, Date startDate, String defaultStatementText, GeneralLedgerBreakdown generalLedgerBreakdown)

Returns TransactionType.

Note that even with the default values, creditType will still be unusable as it will not have the unallocatedGeneralLedger accounts set.



isGeneralLedgerBreakdownValid(GeneralLedgerBreakdown generalLedgerBreakdown)

Returns Boolean.

Call isGlAccountValid() on each of the account numbers in the breakdown.

If any do not pass validation, throw an exception.

For each generalLedgerType, check that one and only one of the percentage values is 0, if not, throw an exception.

For each generalLedgerType, sum the percentage values. If the value is 100 or greater (it cannot be 100) throw an exception.

Otherwise, return true.

setCreditTypePermissableDebit (Long transactionTypeId, PermissableDebit... permissableDebit)

Returns CreditType

Check the transactionType exists, otherwise thrown an exception.

Check that transactionType is a creditType otherwise throw an exception. Change the permissableDebitArray to match the permissableDebit passed in the parameters.

Set editorId to current user, and lastUpdate to current date/time.

setCreditTypeDefaultClearingPeriod (Long transactionTypeId, Int newDefaultClearingPeriod)

Returns CreditType.

Check the transactionType exists, otherwise thrown an exception.

Check that transactionType is a creditType otherwise throw an exception.

Check the default clearing period is valid (0 or greater) otherwise thrown an exception. Set the value.

Set editorId to current user, and lastUpdate to current date/time.

setCreditTypeIsRefundable (Long transactionTypeId, Boolean isRefundable)

Returns CreditType.

Check the transactionType exists, otherwise thrown an exception.

Check that transactionType is a creditType otherwise throw an exception.

Set isRefundable to the new value.

Set editorId to current user, and lastUpdate to current date/time.

setCreditTypeRefundRule (Long transactionTypeId, String refundRule)

Returns CreditType.

Check the transactionType exists, otherwise thrown an exception.

Check that transactionType is a creditType otherwise throw an exception.

Call isRefundRuleValid() with the refund rule. If invalid, thrown an exception.

Otherwise, set the refund rule.

Set editorId to current user, and lastUpdate to current date/time.

setDebitTypeCancellationRule (Long transactionTypeId, String cancellationRule)

Returns DebitType.

Call isCancellationRuleValid() on the cancellationRule. If true, then set debitType.cancellationRule.

setCreditTypeAuthorizationText (Long transactionTypeId, String authorizationText)

Returns CreditType.

Check the transactionType exists, otherwise thrown an exception.

Check that transactionType is a creditType otherwise throw an exception.

Set the new authorization text.

Set editorId to current user, and lastUpdate to current date/time.

setCreditTypeUnallocatedGeneralLedgerAccount (Long transactionTypeId, String unallocatedGeneralLedgerAccount, GlOperation unallocatedGeneralLedgerAccountOperation)

Returns CreditType.

Check the transactionType exists, otherwise thrown an exception.

Check that transactionType is a creditType otherwise throw an exception.

Call isGlAccountValid() on the account number. If false, thrown an exception.

Check that unallocatedGeneralLedgerAccountOperation is either CREDIT or DEBIT, otherwise thrown an exception.

Update the two fields.

Set editorId to current user, and lastUpdate to current date/time.

setTransactionTypeEndDate (Long transactionTypeId, Date endDate)

Returns TransactionType

Check that the transactionType exists, otherwise thrown an exception.

Check the new endDate is after the current startDate, otherwise throw an exception.

Set the new endDate.

Set editorId to current user, and lastUpdate to current date/time.

setTransactionTypeDefaultStatementText (Long transactionTypeId, String defaultStatementText)

Returns TransactionType

Check that the transactionType exists, otherwise thrown an exception.

Set the new defaultStatementText.

Set editorId to current user, and lastUpdate to current date/time.

setTransactionTypeDefaultRollup (Long transactionTypeId, Long rollup)

Returns TransactionType.

Check that the transactionType exists, otherwise thrown an exception.

Check that the rollup exists, otherwise throw an exception.

Set defaultRollup to the new rollup.

Set editorId to current user, and lastUpdate to current date/time.

addTagToTransactionType (Long transactionTypeId, Tag tag)

Returns TransactionType.

Check that the transactionType exists, otherwise thrown an exception.

Check that the tag exists, otherwise throw an exception.

Check that the tag doesn't exist in the current tagArray [] otherwise throw an exception.

Add tag to the tagArray[]

Set editorId to current user, and lastUpdate to current date/time.

removeTagFromTransactionType (Long transactionTypeId, Tag tag)

Returns TransactionType.

Check that the transactionType exists, otherwise thrown an exception.

Check that the tag exists, otherwise throw an exception.

Check that the tag exists in the current tagArray [] otherwise throw an exception.

Remove tag from the tagArray[]

Set editorId to current user, and lastUpdate to current date/time.

addNewGeneralLedgerBreakdownToTransactionType (Long transactionTypeId, GeneralLedgerBreakdown generalLedgerBreakdown)

Returns TransactionType.

Check that the transactionType exists, otherwise thrown an exception.

Call checkGeneralLedgerBreakdown() on the passed generalLedgerBreakdown. If there is a problem, pass the exception back to the caller.

Overwrite the current generalLedgerBreakdown with the passed generalLedgerBreakdown.

persistTransactionType (transactionType)

Returns TransactionType

Check that there is a valid start date.

If endDate is set, check that it is after startDate.

Check defaultStatementText is not null.

Call checkGeneralLedgerBreakdown() with the generalLedgerBreakdown.

If this is a debitType, call isCancellationRuleValid()

If this is a creditType, call isRefundRuleValid()

If this is a creditType, check that both unallocatedGeneralLedgerAccount and unallocatedGeneralLedgerAccountOperation are not null.

If this is a creditType, call isGlAccountValid() on the unallocatedGeneralLedgerAccount.

If any of these checks fail, thrown an exception.

If all these checks pass:

If creatorId is null, set creatorId to the current user.

If creatorId is already set, set editorId to the current user.

Set lastUpdate to the current date/time.

Persist the transaction in the database.

User Preference Service [UserPreferenceService]

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.

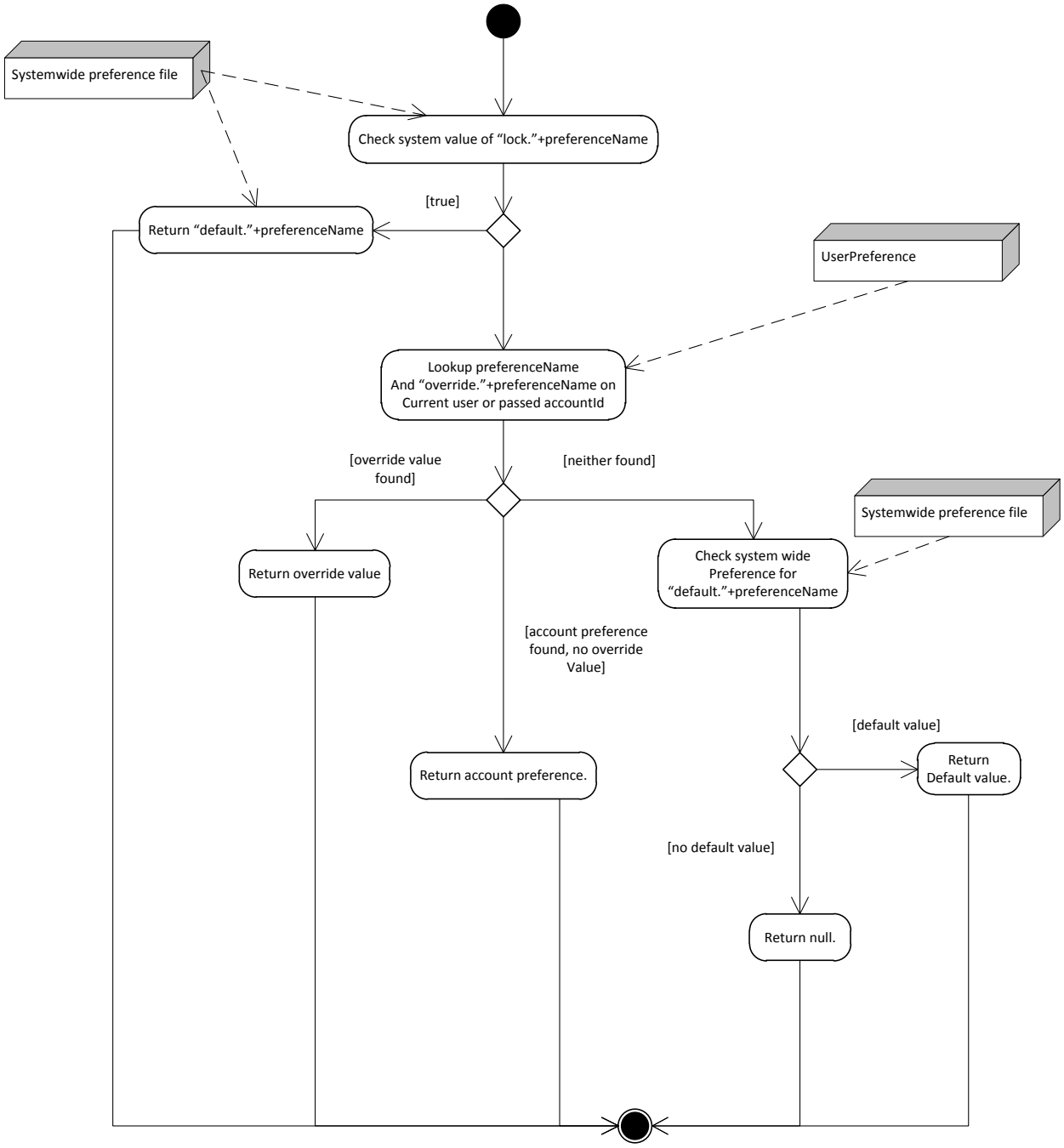
If a school does not want a student to be able to set refund method, then locked.refund.method will be set to true. In this case, default.refund.method will apply no matter what values might be stored in user preferences.

getUserPreferences (String userId)

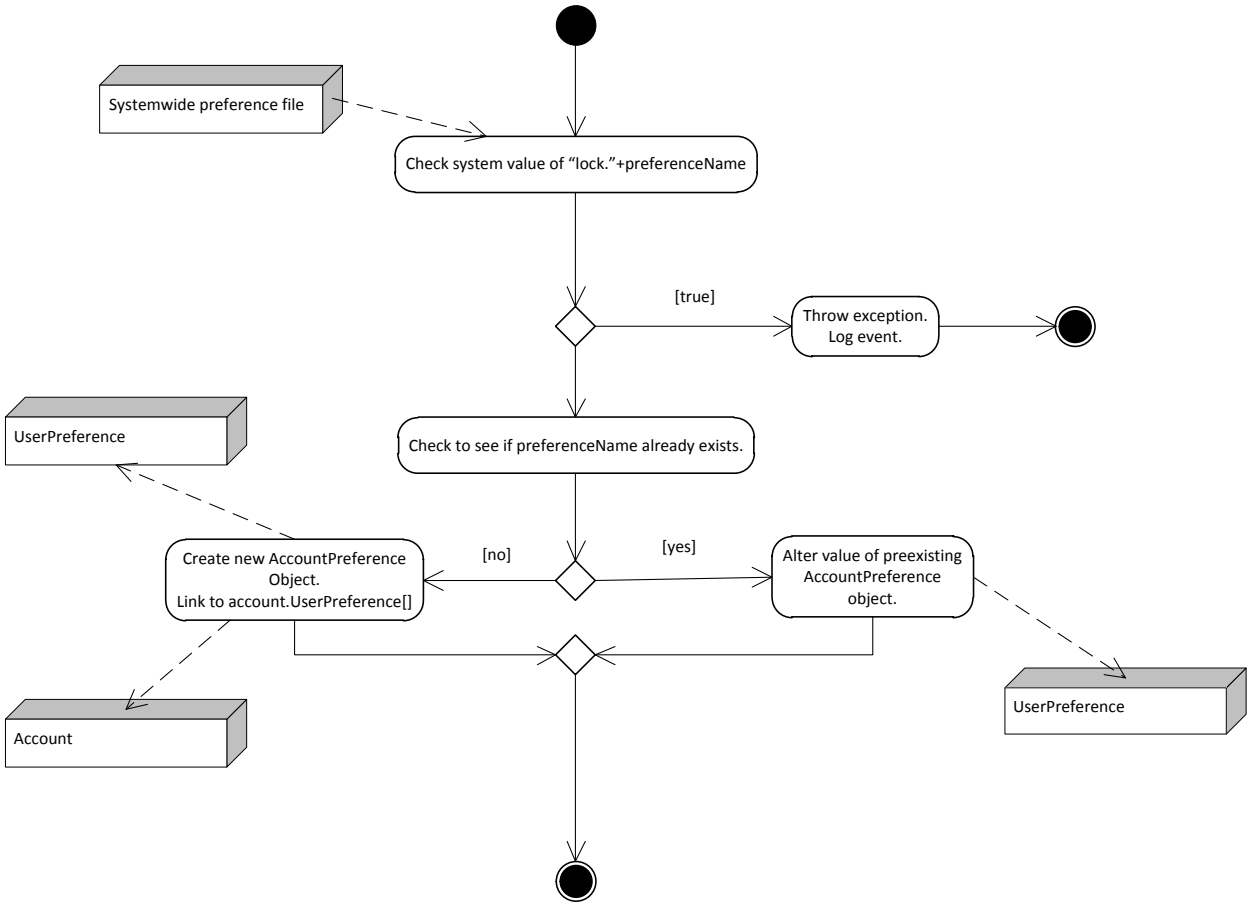
Returns List<UserPreference>

getPreference (preferenceName)

getPreference (preferenceName, accountId)



setPreference (preferenceName)
setPreference (preferenceName, accountId)



Transfer Transaction Sub Service (In progress) [TransactionService]

The transfer transaction service is a set of methods that take care of transferring transactions from one account to another, or, in the case of payment billing, within the account, breaking a group of transactions into payments with different due dates.

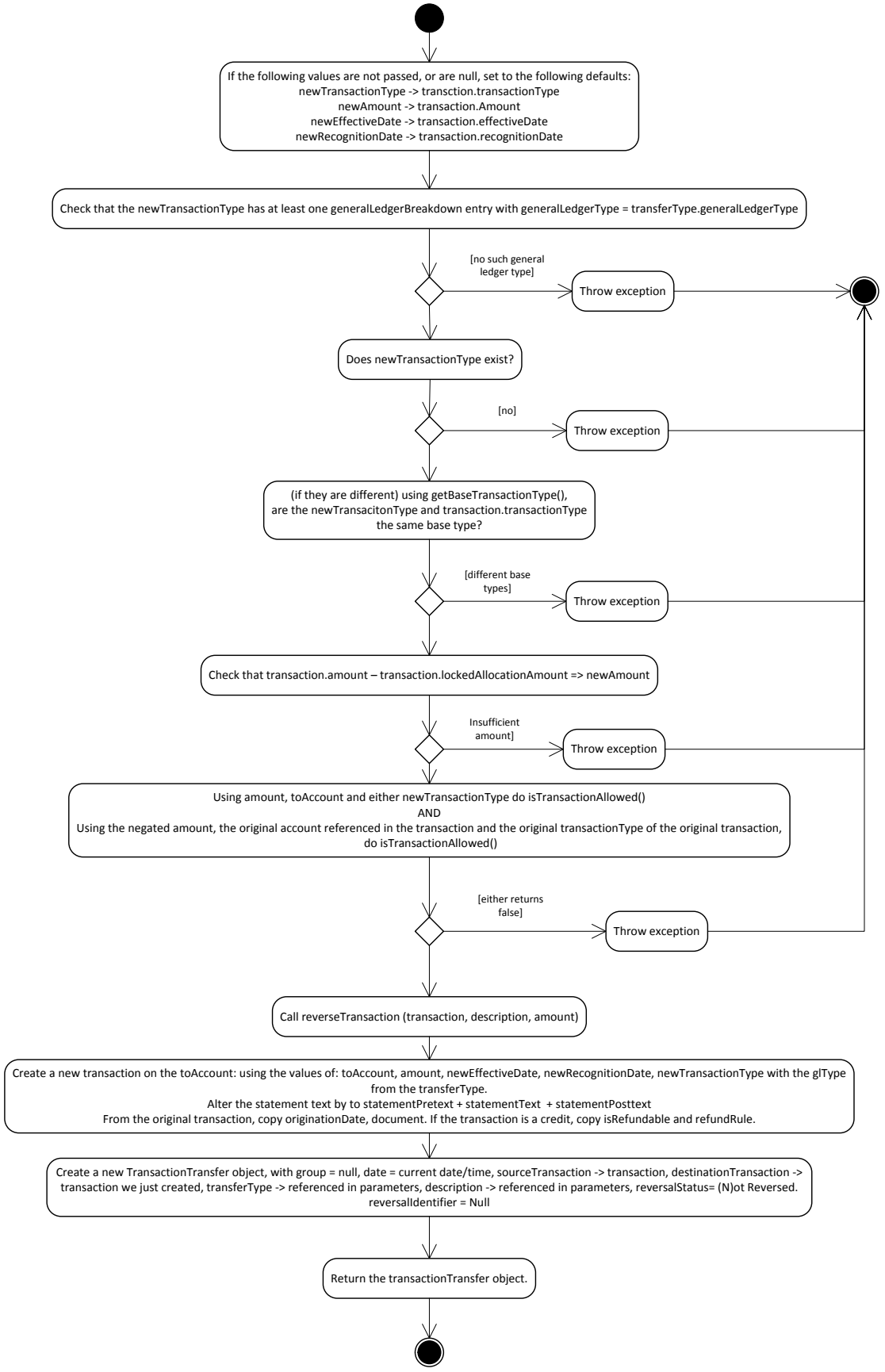
Transfer transaction is a method that transfers responsibility for a transaction from one account to another. It does this by issuing a negative transaction on the original account to wipe out its value (and its general ledger effect, if appropriate) which is done via the reverseTransaction() method.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

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.

transferTransaction (Long transactionId, String toAccount, TransferType transferType, String statementPretext, String statementPosttext)

transferTransaction (Long transactionId, String toAccount, TransferType transferType, String statementPretext, String statementPosttext, BigDecimal amount, TransactionType newTransactionType, Date newEffectiveDate, Date newRecognitionDate)

Returns TransactionTransfer.


If the following values are not passed, or are null, set to the following defaults:
newTransactionType -> transction.transactionType
newAmount -> transaction.Amount
newEffectiveDate -> transaction.effectiveDate
newRecognitionDate -> transaction.recognitionDate
Check that the newTransactionType has at least one generalLedgerBreakdown entry with generalLedgerType = transferType.generalLedgerType
[no such general ledger type]
Throw exception
Does newTransactionType exist?
[no]
Throw exception
(if they are different) using getBaseTransactionType(), are the newTransacitonType and transaction.transactionType the same base type?
[different base types]
Throw exception
Check that transaction.amount – transaction.lockedAllocationAmount => newAmount
Insufficient amount]
Throw exception
Using amount, toAccount and either newTransactionType do isTransactionAllowed()
AND
Using the negated amount, the original account referenced in the transaction and the original transactionType of the original transaction, do isTransactionAllowed()
[either returns false]
Throw exception
Call reverseTransaction (transaction, description, amount)
Create a new transaction on the toAccount: using the values of: toAccount, amount, newEffectiveDate, newRecognitionDate, newTransactionType with the glType from the transferType.
Alter the statement text by to statementPretext + statementText + statementPosttext
From the original transaction, copy originationDate, document. If the transaction is a credit, copy isRefundable and refundRule.
Create a new TransactionTransfer object, with group = null, date = current date/time, sourceTransaction -> transaction, destinationTransaction -> transaction we just created, transferType -> referenced in parameters, description -> referenced in parameters, reversalStatus= (N)ot Reversed. reversalIdentifier = Null
Return the transactionTransfer object.

persistTransactionTransfer (TransactionTransfer transactionTransfer)

Returns TransactionTransfer.

If the transactionTransfer object has a non-null transaction group, check if any other transactionTransfer objects exist with the same transferGroup, check that the account referenced in sourceTransaction and the account referenced in destinationTransaction match the accounts referenced in this transasctionTransfer's sourceTransaction.account and destinationTransaction.account. If not, throw an exception.

Otherwise, persist the transactionTransfer object.

setTransferTransactionGroupRollup (String transferGroup, Rollup rollup)

Returns void.

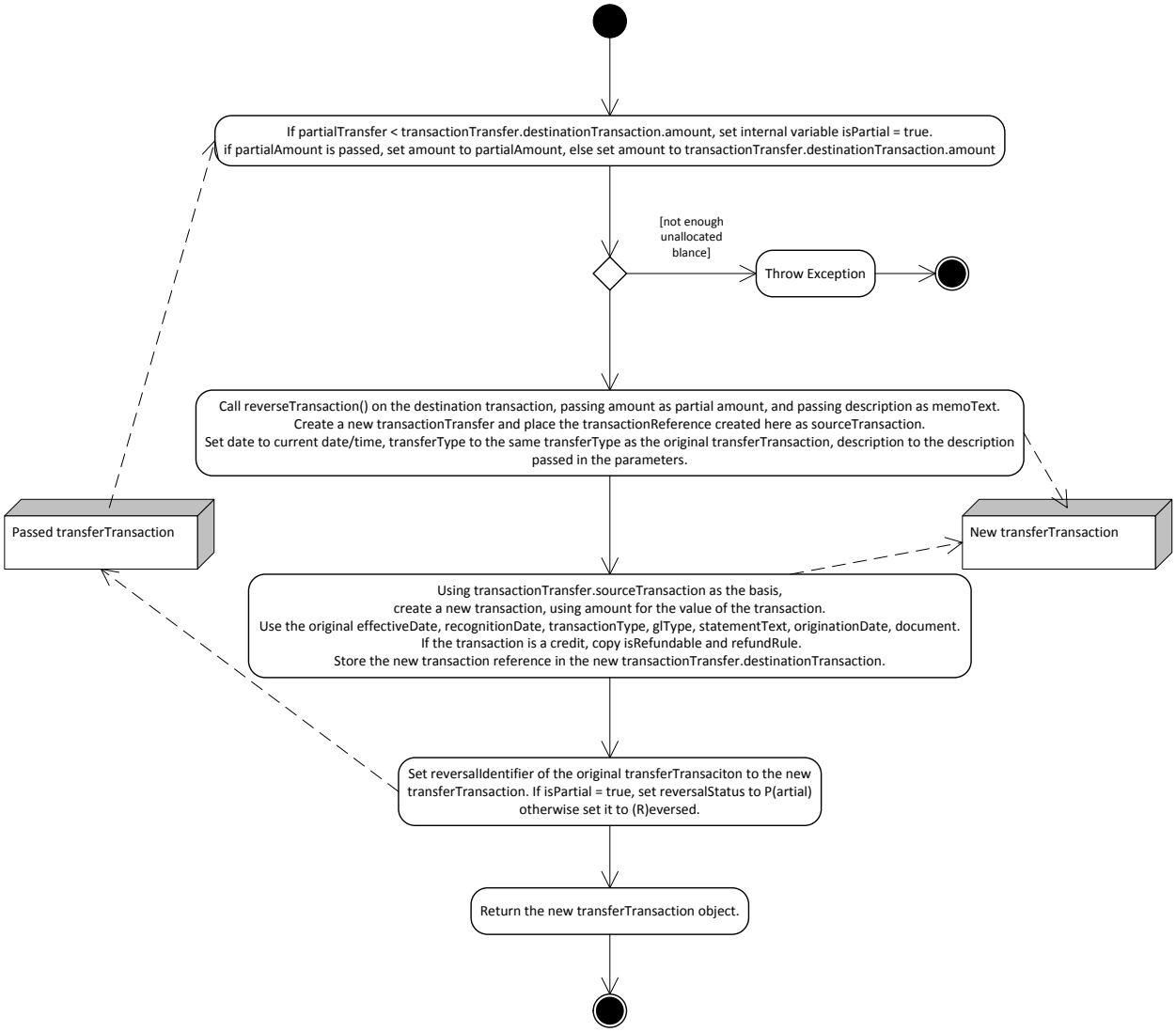
Iterate through every destinationTransaction of all TransferTransction objects that have groupIdentifier as their transferGroup. Set the rollup of each of the destination transactions to the passed rollup.

reverseTransferTransaction (TransactionTransfer transactionTransfer, String description)

reverseTransferTransaction (TransactionTransfer transactionTransfer, String description, BigDecimal partialAmount)

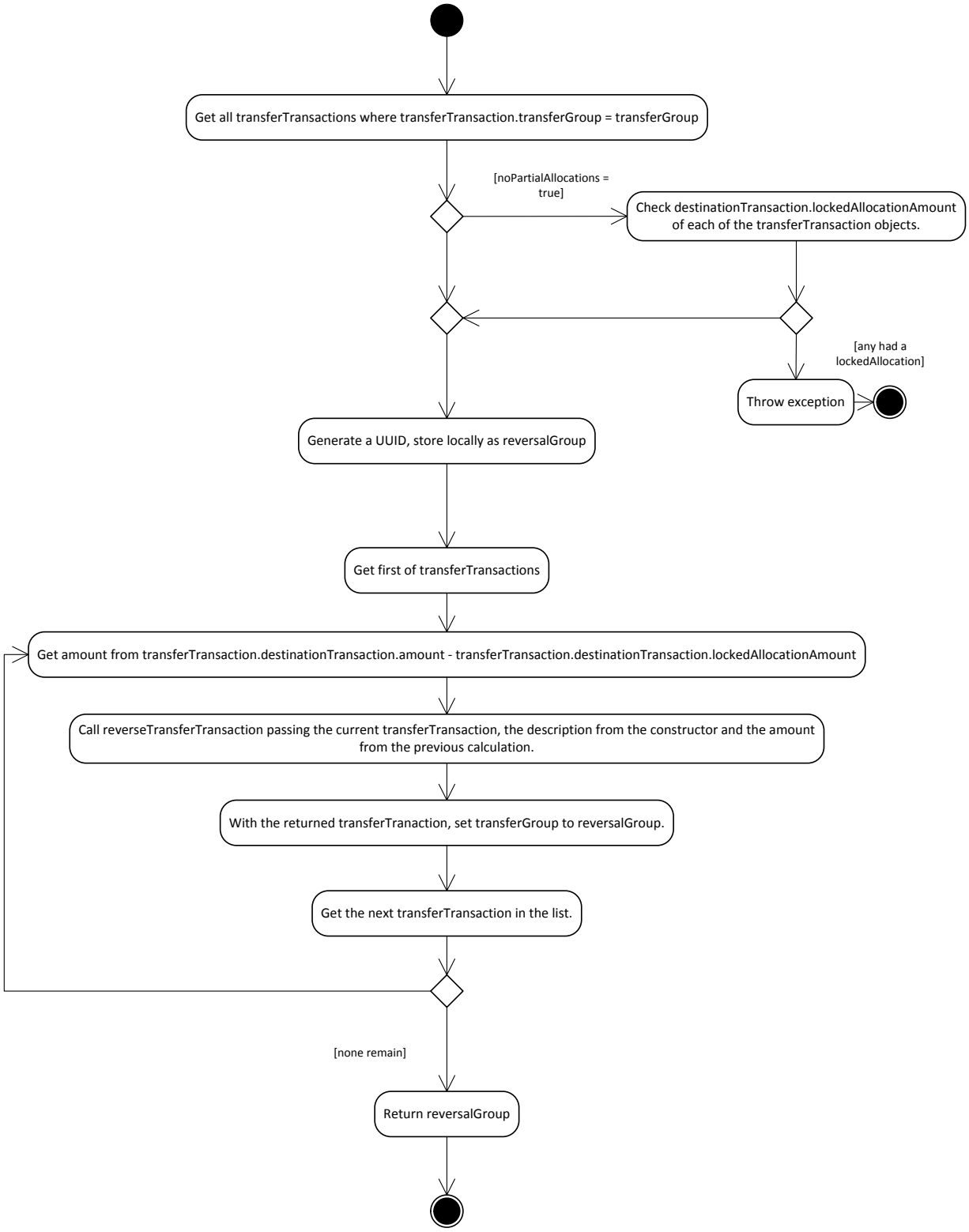
This method can be used to reverse a previously made transaction transfer. This allows the system to dynamically restore the charges and credits to an originating account without the user having to look up those values.

Returns TransactionTransfer



reverseTransferTransactionGroup (String transferTransactionGroup, String description, Boolean noPartialAllocations)

Returns List<TransactionTransfer>


Get all transferTransactions where transferTransaction.transferGroup = transferGroup
[noPartialAllocations = true]
Check destinationTransaction.lockedAllocationAmount of each of the transferTransaction objects.
[any had a lockedAllocation]
Throw exception
Generate a UUID, store locally as reversalGroup
Get first of transferTransactions
Get amount from transferTransaction.destinationTransaction.amount - transferTransaction.destinationTransaction.lockedAllocationAmount
Call reverseTransferTransaction passing the current transferTransaction, the description from the constructor and the amount from the previous calculation.
With the returned transferTranaction, set transferGroup to reversalGroup.
Get the next transferTransaction in the list.
[none remain]
Return reversalGroup

Auditable Entities Services

KSA relies on a number of classes that are derived from the AuditableEntity class. These methods permit the creation and editing of these types.

persistRollup (Rollup rollup)

Returns Rollup.

Check that no other rollup exists with the same Code (and id is different). If it does, throw exception.

If this is a new rollup, populate the id field, set creatorId to the current user, and creationDate to the current system date/time. and then persist.

If this is an update, set the editorId to the current user, and the lastUpdated to the current date/time, then persist.

persistTag (Tag tag)

Returns Tag.

Check that no other tag exists with the same Code (and id is different). If it does, throw exception.

If this is a new tag, populate the id field, set creatorId to the current user, and creationDate to the current system date/time. And then persist.

If this is an update, set the editorId to the current user, and the lastUpdated to the current date/time, then persist.

persistBankType (BankType bankType)

Returns BankType.

Check that no other bankType exists with the same Code (and id is different). If it does, throw exception.

If this is a new bankType, populate the id field, set creatorId to the current user, and creationDate to the current system date/time. and then persist.

If this is an update, set the editorId to the current user, and the lastUpdated to the current date/time, then persist.

persistIdentityType (IdentityType identityType)

Returns IdentityType.

Check that no other identityType exists with the same Code (and id is different). If it does, throw exception.

If this is a new identityType, populate the id field, set creatorId to the current user, and creationDate to the current system date/time. And then persist.

If this is an update, set the editorId to the current user, and the lastUpdated to the current date/time, then persist.

persistTaxType (TaxType taxType)

Returns TaxType.

Check that no other taxType exists with the same Code (and id is different). If it does, throw exception.

If this is a new taxType, populate the id field, set creatorId to the current user, and creationDate to the current system date/time. and then persist.

If this is an update, set the editorId to the current user, and the lastUpdated to the current date/time, then persist.

persistAccountStatusType (AccountStatusType accountStatusType)

Returns AccountStatusType.

Check that no other accountStatusType exists with the same Code (and id is different). If it does, throw exception.

If this is a new accountStatusType, populate the id field, set creatorId to the current user, and creationDate to the current system date/time. and then persist.

If this is an update, set the editorId to the current user, and the lastUpdated to the current date/time, then persist.

persistAccountType (AccountType accountType)

Returns AccountType.

Check that no other accountType exists with the same Code (and id is different). If it does, throw exception.

If this is a new accountType, populate the id field, set creatorId to the current user, and creationDate to the current system date/time. and then persist.

If this is an update, set the editorId to the current user, and the lastUpdated to the current date/time, then persist.

persistGeneralLedgerType (GeneralLedgerType generalLedgerType)

Returns GeneralLedgerType.

Check that no other generalLedgerType exists with the same Code (and id is different). If it does, throw exception.

If this is a new accountStatusType, populate the id field, set creatorId to the current user, and creationDate to the current system date/time. and then persist.

If this is an update, set the editorId to the current user, and the lastUpdated to the current date/time, then persist.

persistActivityType (ActivityType activityType)

Returns ActivityType.

Check that no other activityType exists with the same Code (and id is different). If it does, throw exception.

If this is a new activityType, populate the id field, set creatorId to the current user, and creationDate to the current system date/time. and then persist.

If this is an update, set the editorId to the current user, and the lastUpdated to the current date/time, then persist.

persistRefundType (RefundType refundType)

Returns RefundType.

Check that no other refundType exists with the same Code (and id is different). If it does, throw exception.

If this is a new refundType, populate the id field, set creatorId to the current user, and creationDate to the current system date/time. and then persist.

If this is an update, set the editorId to the current user, and the lastUpdated to the current date/time, then persist.

persistTransactionTypeMaskRole (TransactionTypeMaskRole transactionTypeMaskRole)

Returns TransactionTypeMaskRole.

Check that no other transactionType exists with the same Code (and id is different). If it does, throw exception.

Check the transaction mask is a valid regular expression. If not, throw exception.

Check the role exists with KIM, otherwise throw exception.

If this is a new transferType, populate the id field, set creatorId to the current user, and creationDate to the current system date/time. and then persist.

If this is an update, set the editorId to the current user, and the lastUpdated to the current date/time, then persist.

persistTransferType (TransferType transferType)

Returns RefundType.

Check that no other refundType exists with the same Code (and id is different). If it does, throw exception.

Check that generalLedgerType is valid, if not, throw exception.

If this is a new refundType, populate the id field, set creatorId to the current user, and creationDate to the current system date/time. and then persist.

If this is an update, set the editorId to the current user, and the lastUpdated to the current date/time, then persist.

Refund Service

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.

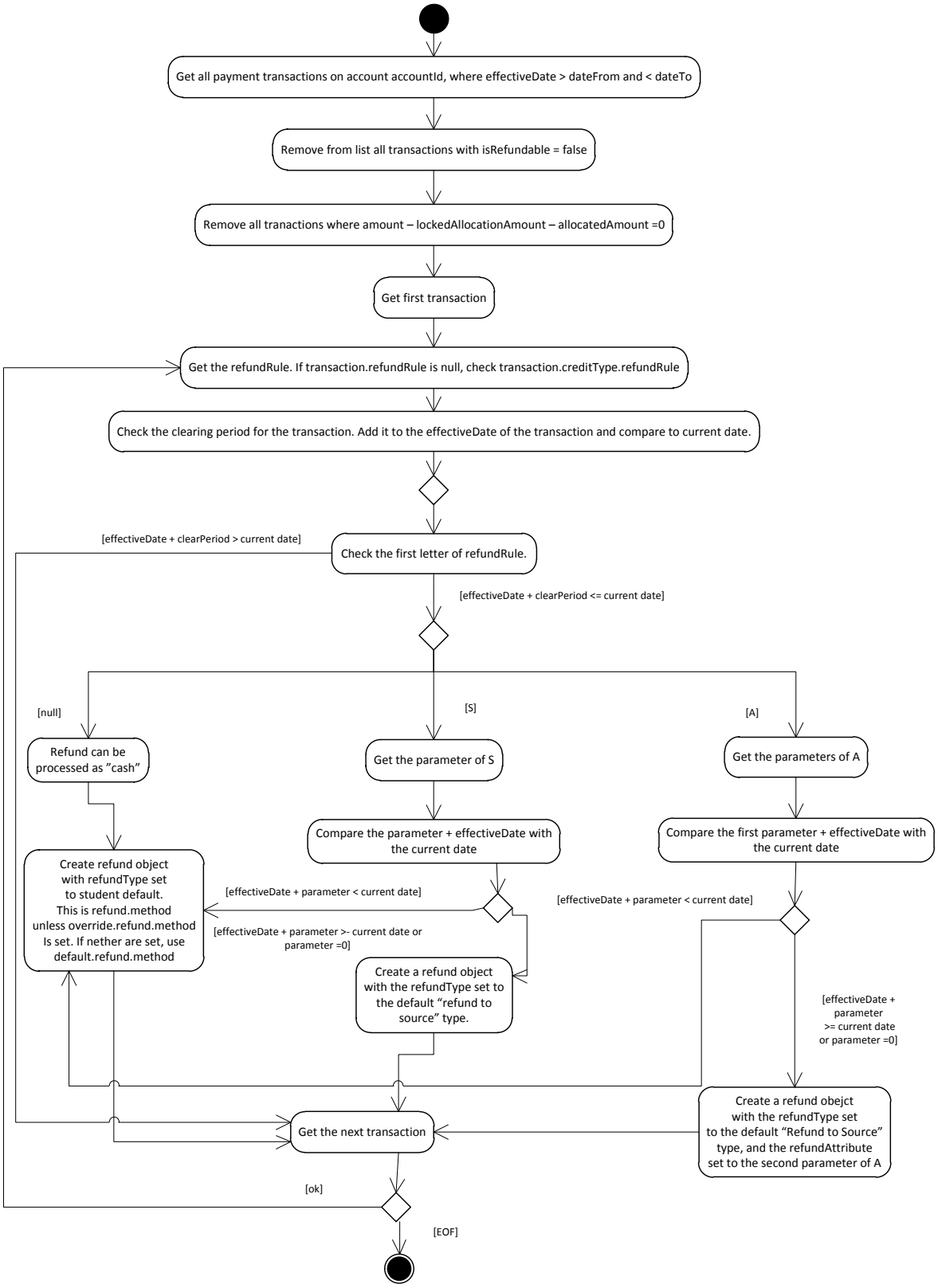
The refund service will try to determine if a transaction can be paid as “cash” (that is, without restrictions, not going to a third party, or being repaid in a specific way.) If a refund (or part of a refund) can be paid as “cash” then it will be paid with the refund type as defined in refund.method, 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 (String accountId, Date fromDate, Date toDate)

Returns List<Refund>

This creates a list of unverified refunds.



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 (List<String accountId>, Date fromDate, Date toDate)

Returns List<Refund>

For each accountId, perform a checkForRefund()

checkForRefunds ()

checkForRefunds (Date dateFrom, Date dateTo)

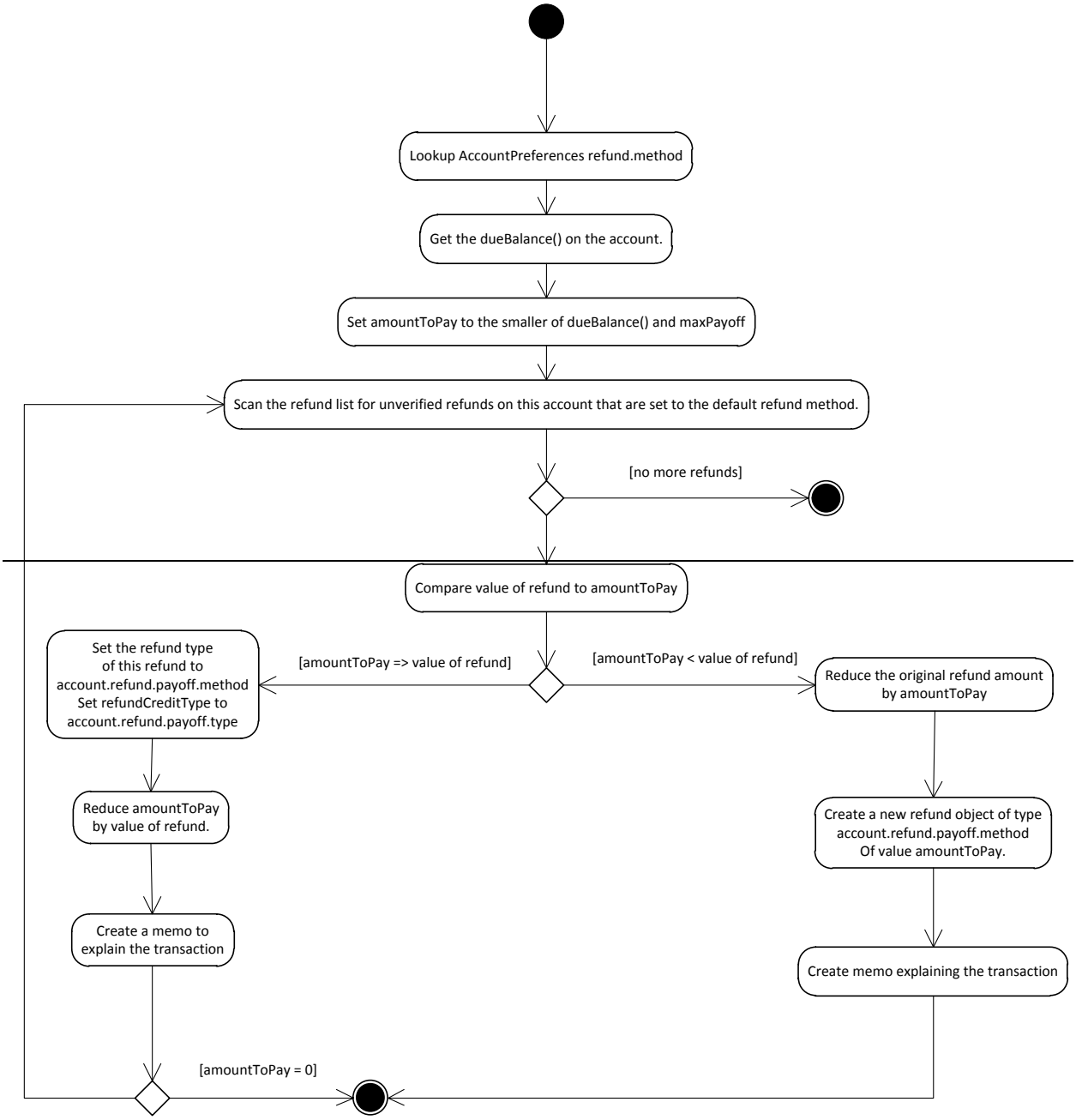
Returns List<Refund>

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)~~

Ignore while the specifics of this are looked into.

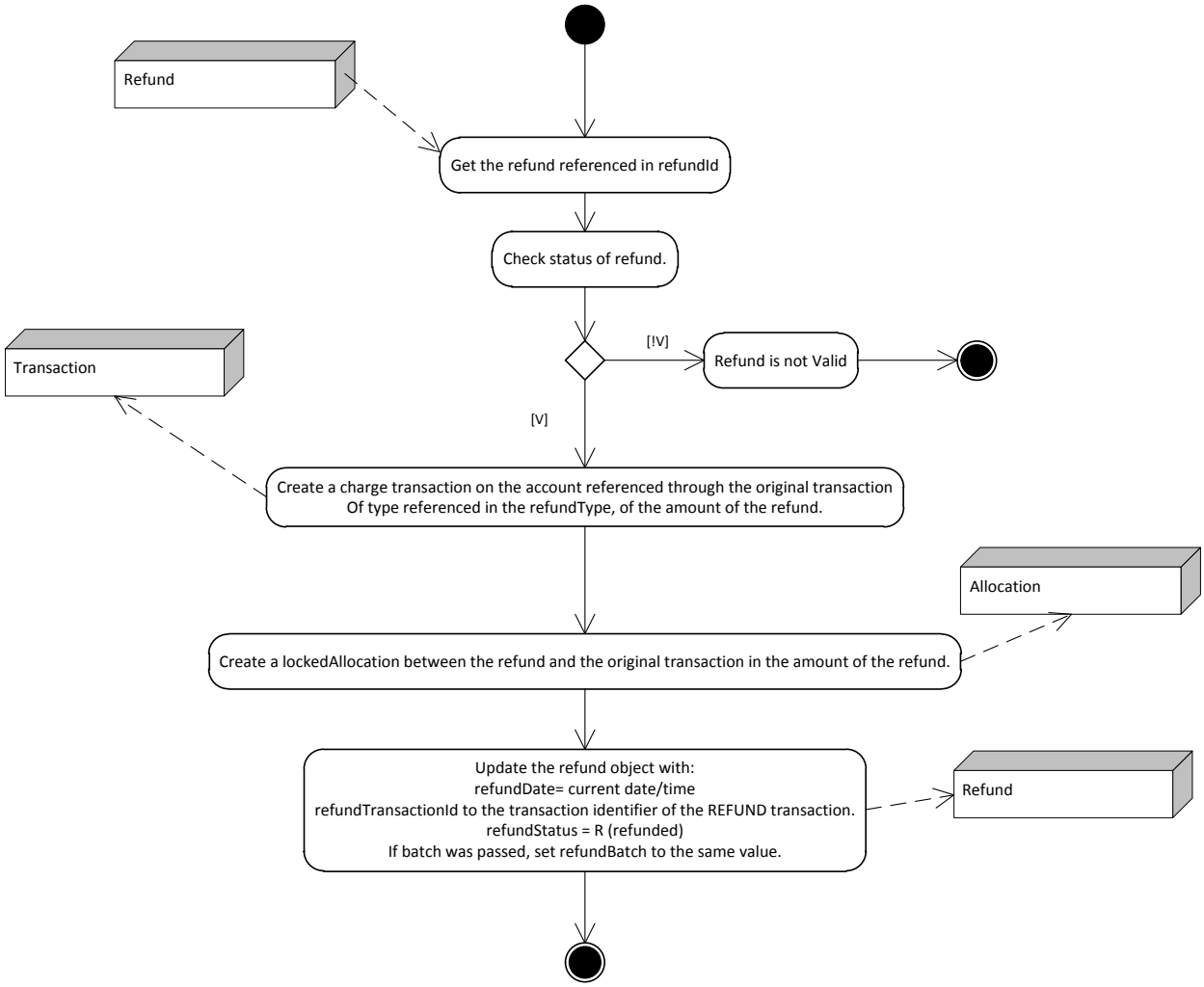


performRefund(Long refundId)

performRefund(Long refundId, String batch)

Returns Refund.

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



validateRefund (Long refundId)

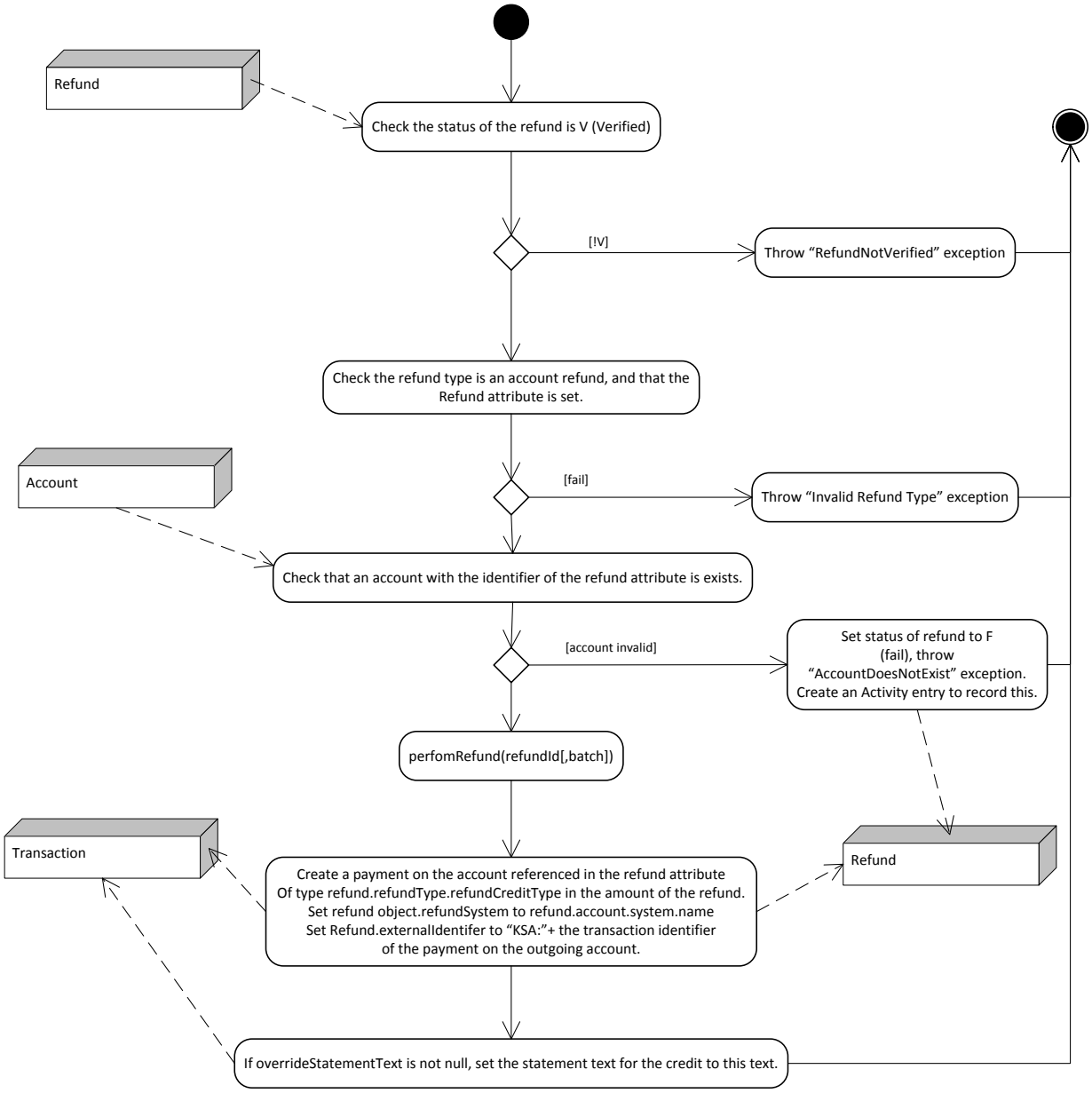
Returns Refund.

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

doAccountRefund (Long refundId)

doAccountRefund (Long refundId, String batch)

Returns Refund.



doAccountRefunds (String batch)

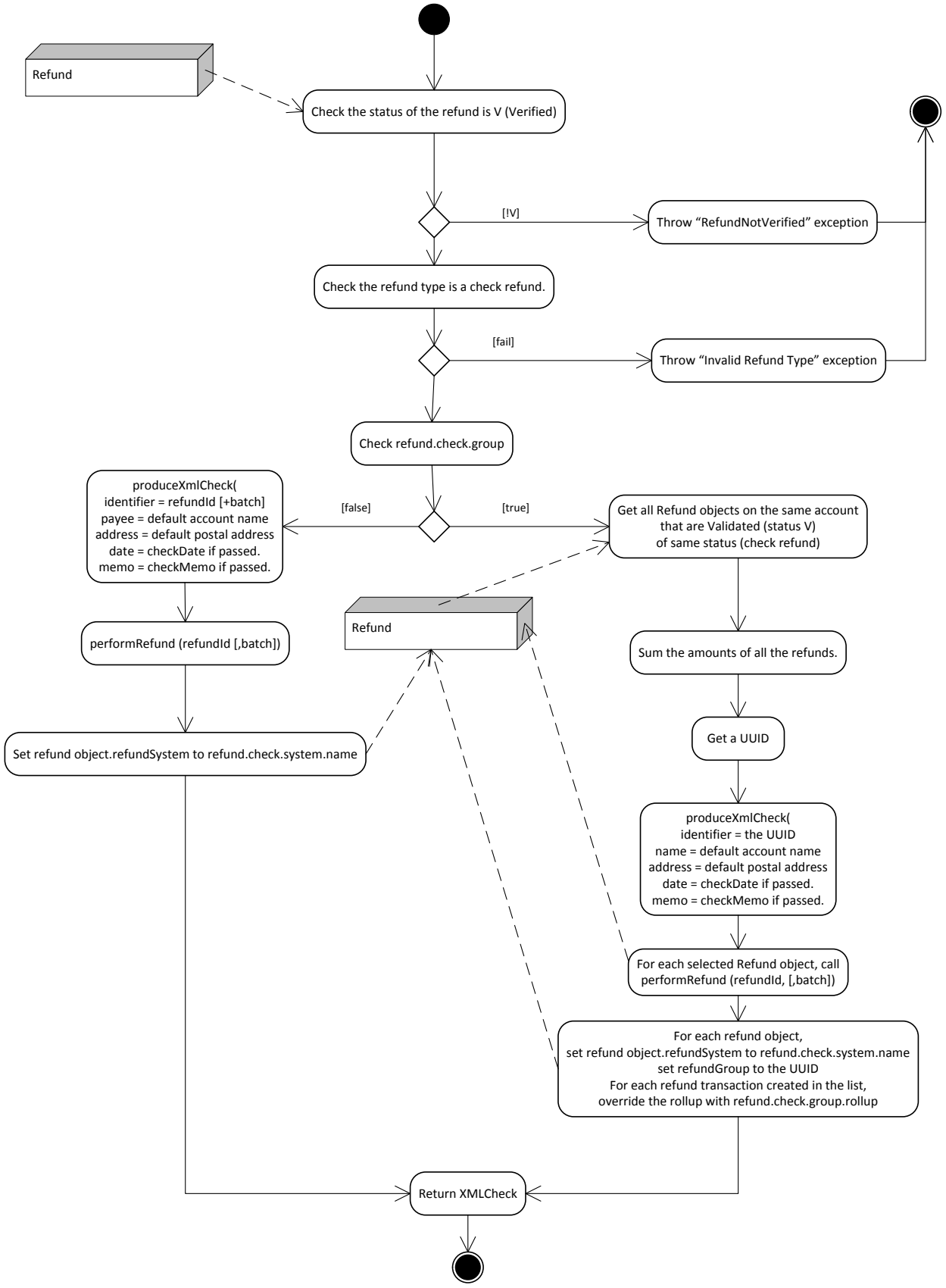
Returns List<Refund>

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

doCheckRefund(Long refundId, Date checkDate, String checkMemo)

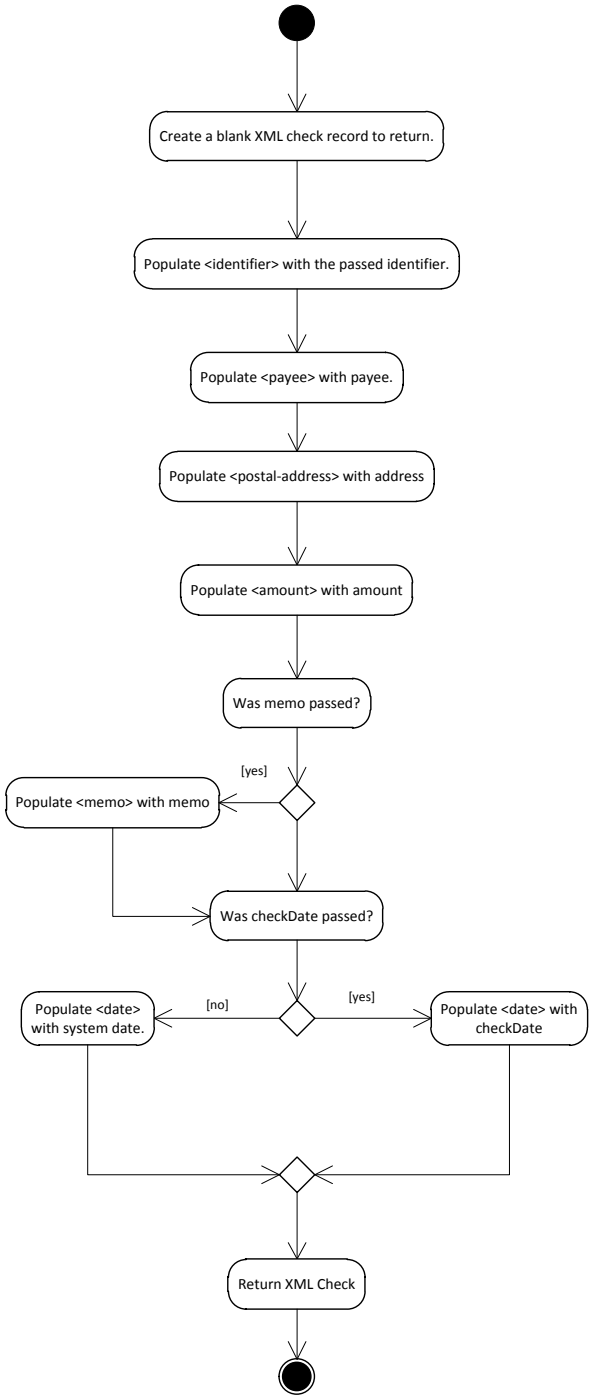
doCheckRefund(Long refundId, Date checkDate, String checkMemo, String batch)

Returns String (XmlCheck).


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
produceXmlCheck(
identifier = refundId [+batch]
payee = default account name
address = default postal address
date = checkDate if passed.
memo = checkMemo if passed.
[false]
[true]
Get all Refund objects on the same account
that are Validated (status V)
of same status (check refund)
performRefund (refundId [,batch])
Refund
Sum the amounts of all the refunds.
Get a UUID
Set refund object.refundSystem to refund.check.system.name
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.check.group.rollup
Return XMLCheck

produceXmlCheck (String identifier, String payee, PostalAddress postalAddress, BigDecimal amount, String memo, Date checkDate)

Returns String (XmlCheck).



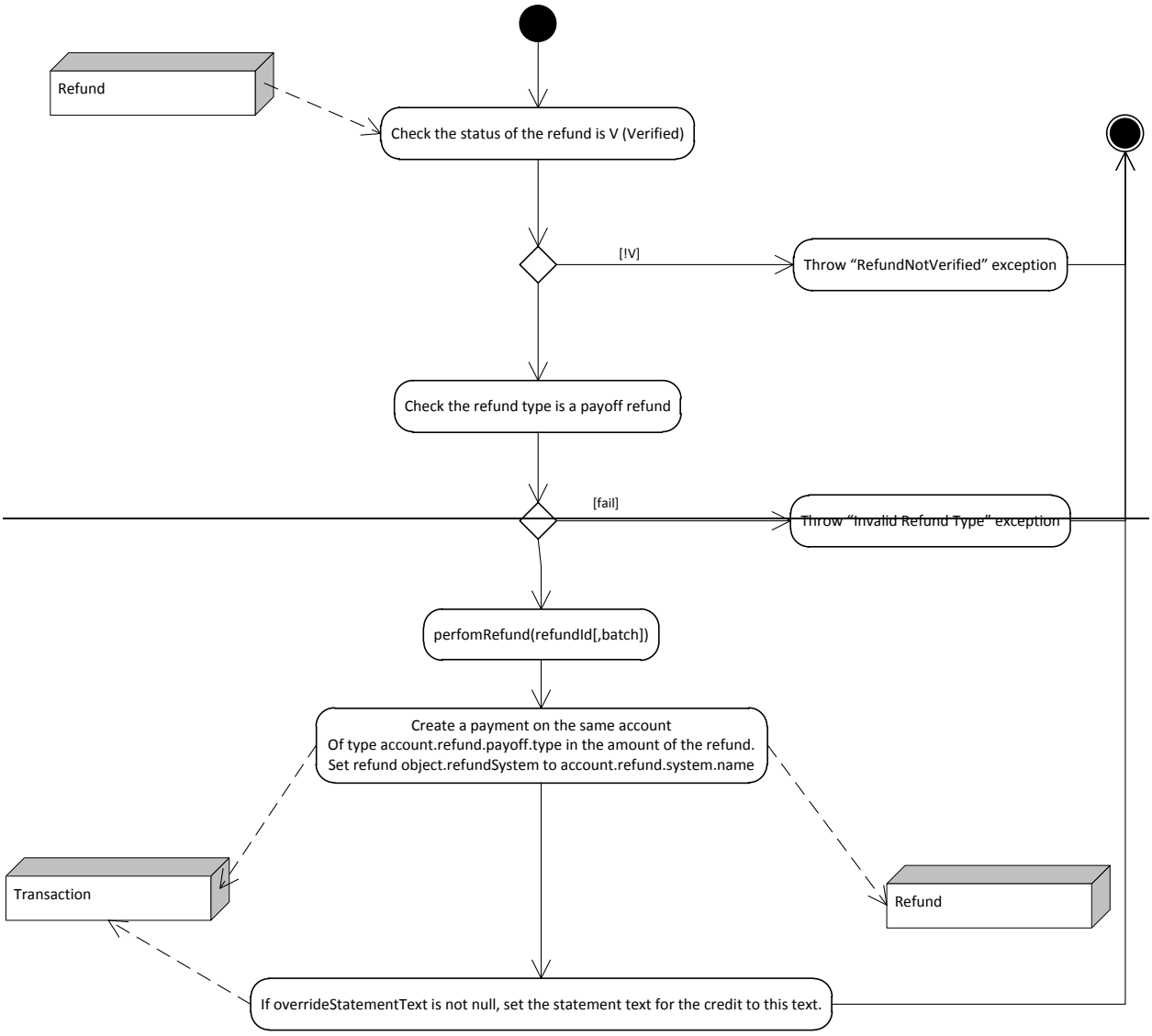
doCheckRefunds(String batch, String checkMemo, Date checkDate)

Returns String (batch-check).

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)~~



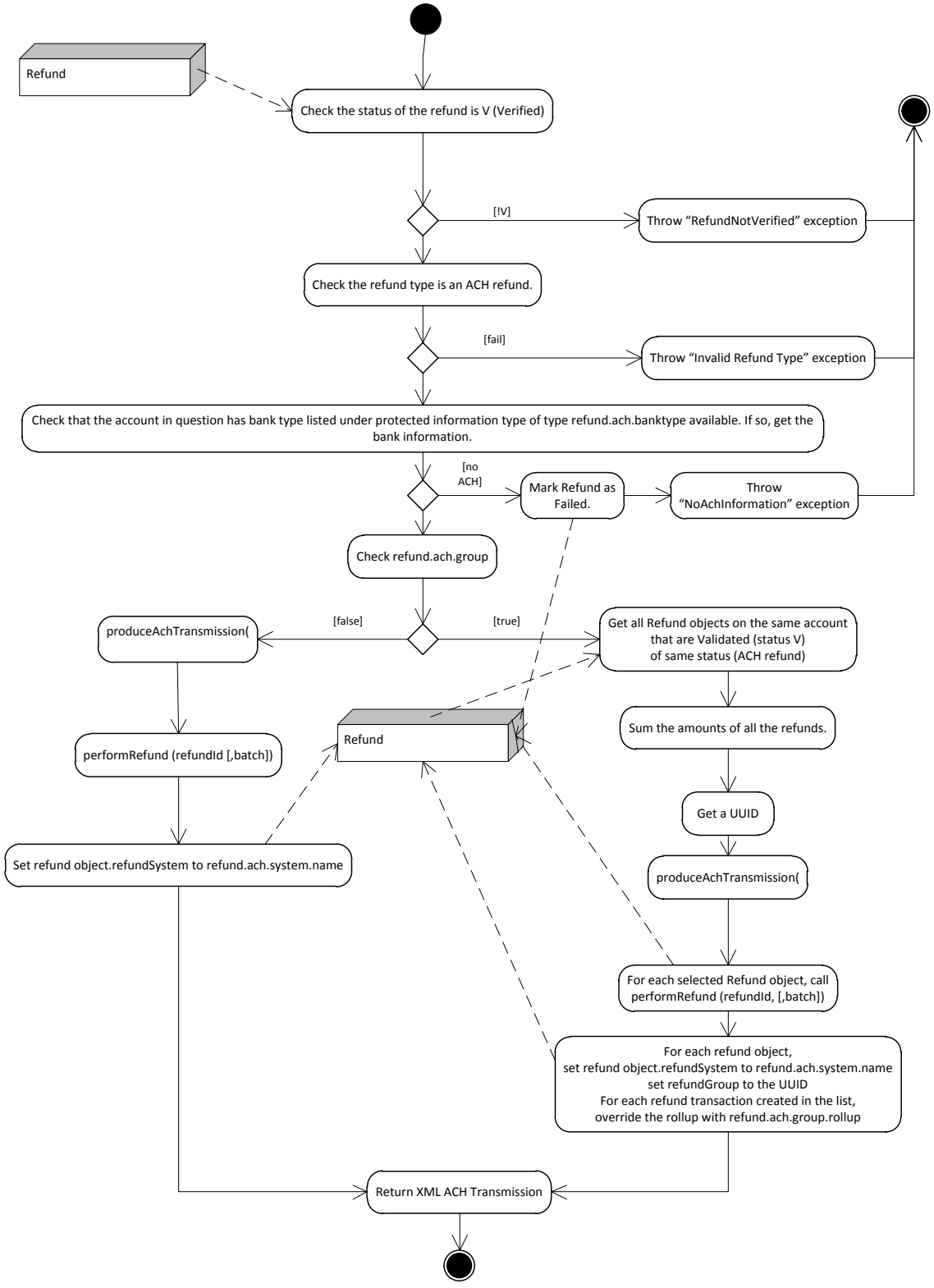
~~doAllPayoffRefunds (batch)~~

~~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(Long refundId)

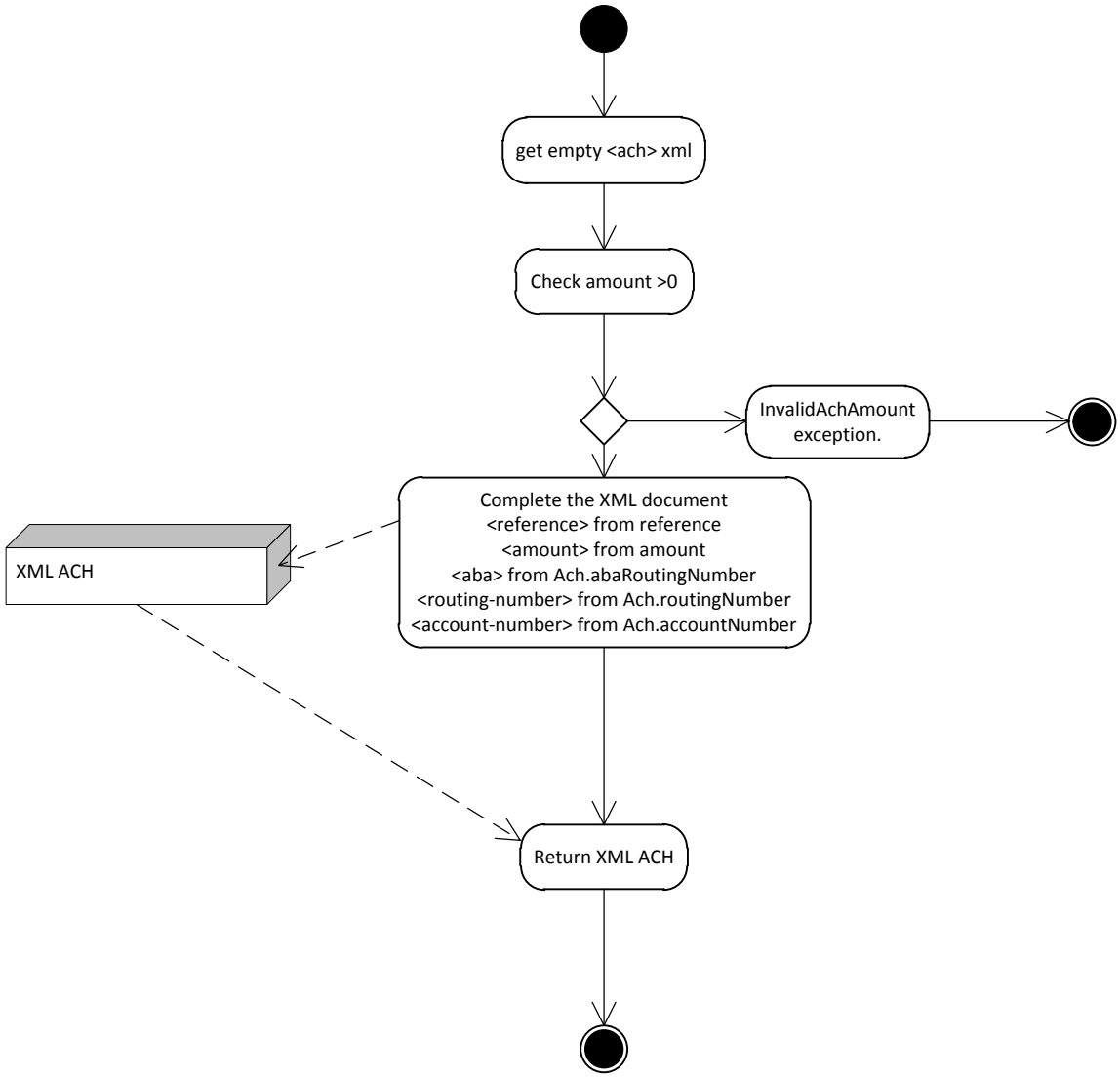
doAchRefund(Long refundId, String batch)

Return String (XmlAch).



produceAchTransmission (BigDecimal amount, String reference, Ach ach)

Returns String (ach).



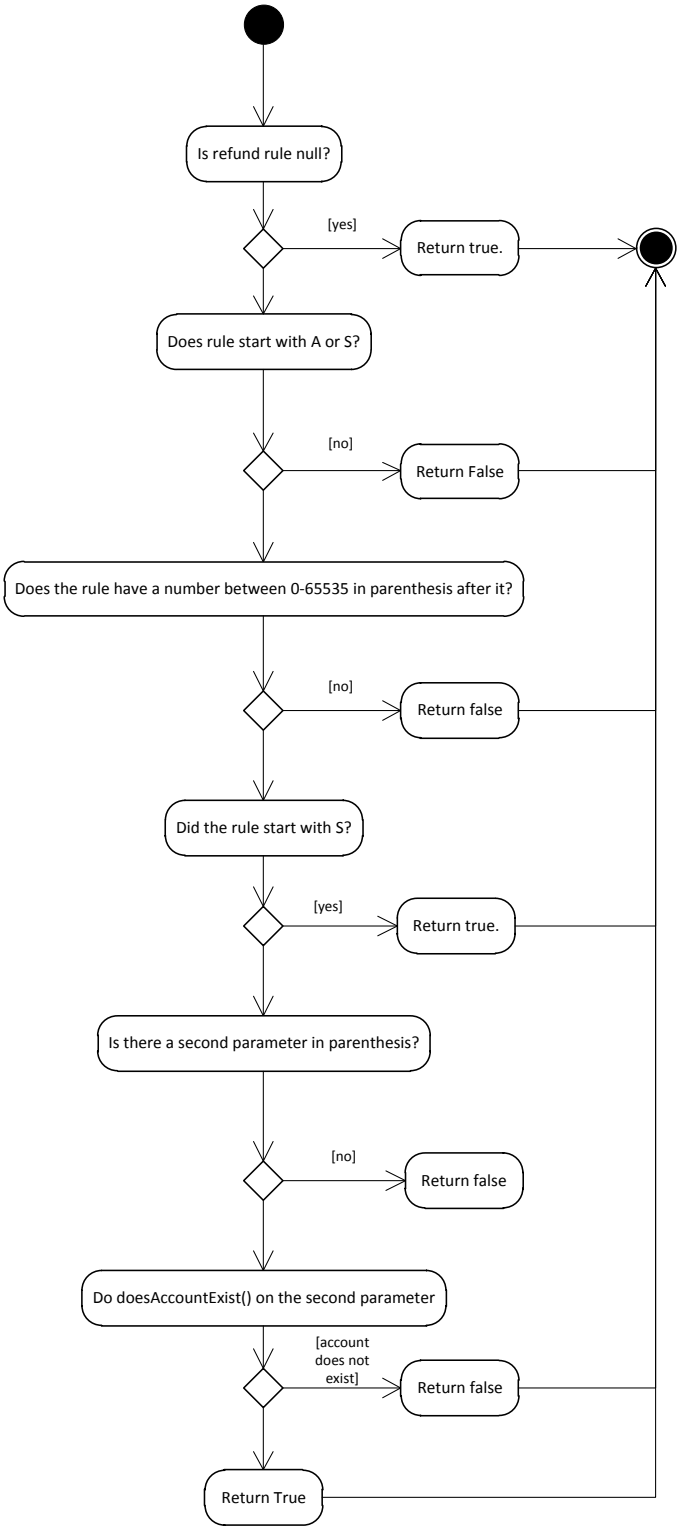
doAchRefunds (String batch)

Return String (batch-ach)

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).

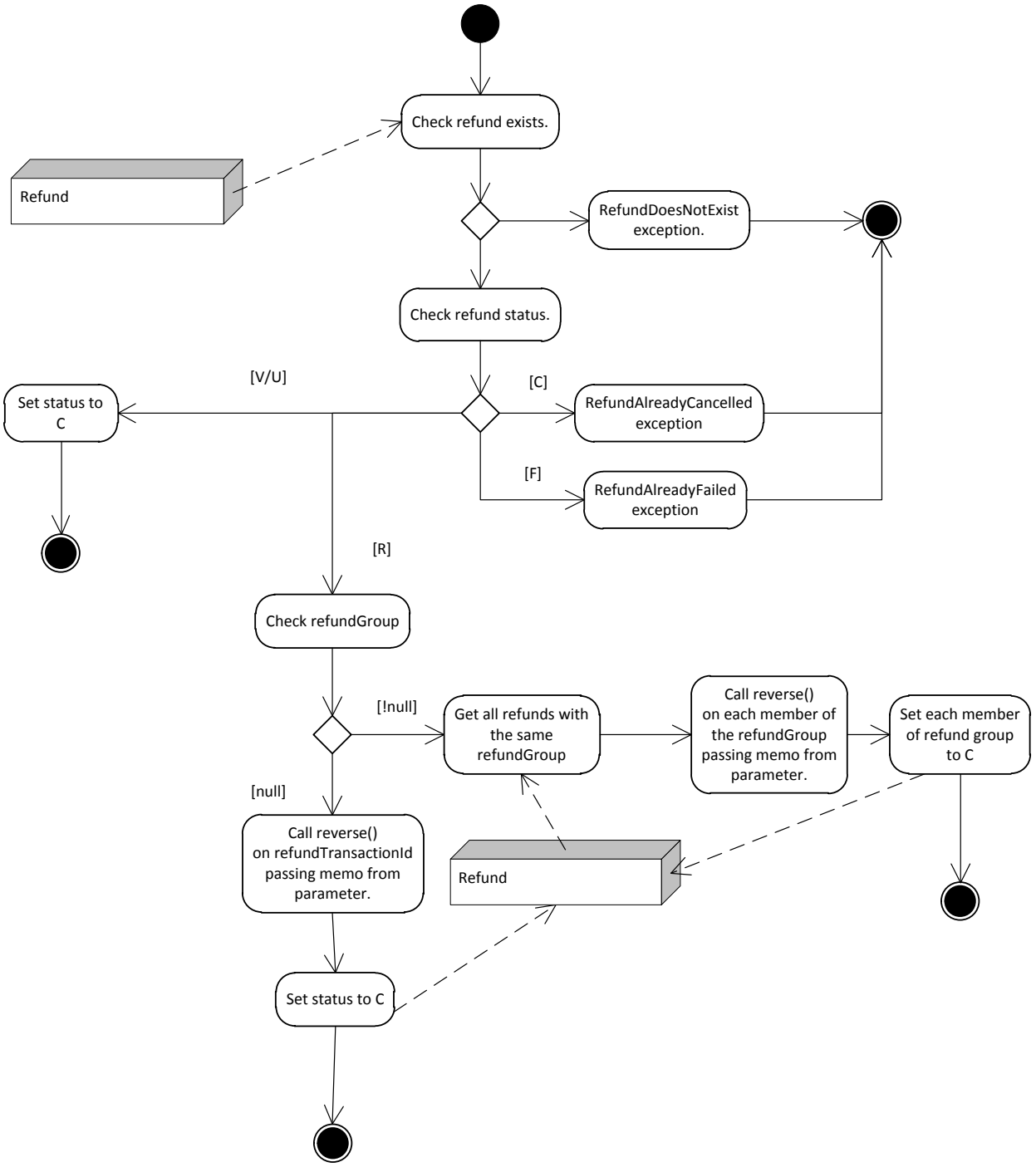
isRefundRuleValid(String refundRule)

Returns Boolean.



cancelRefund (Long refundId, String memoText)

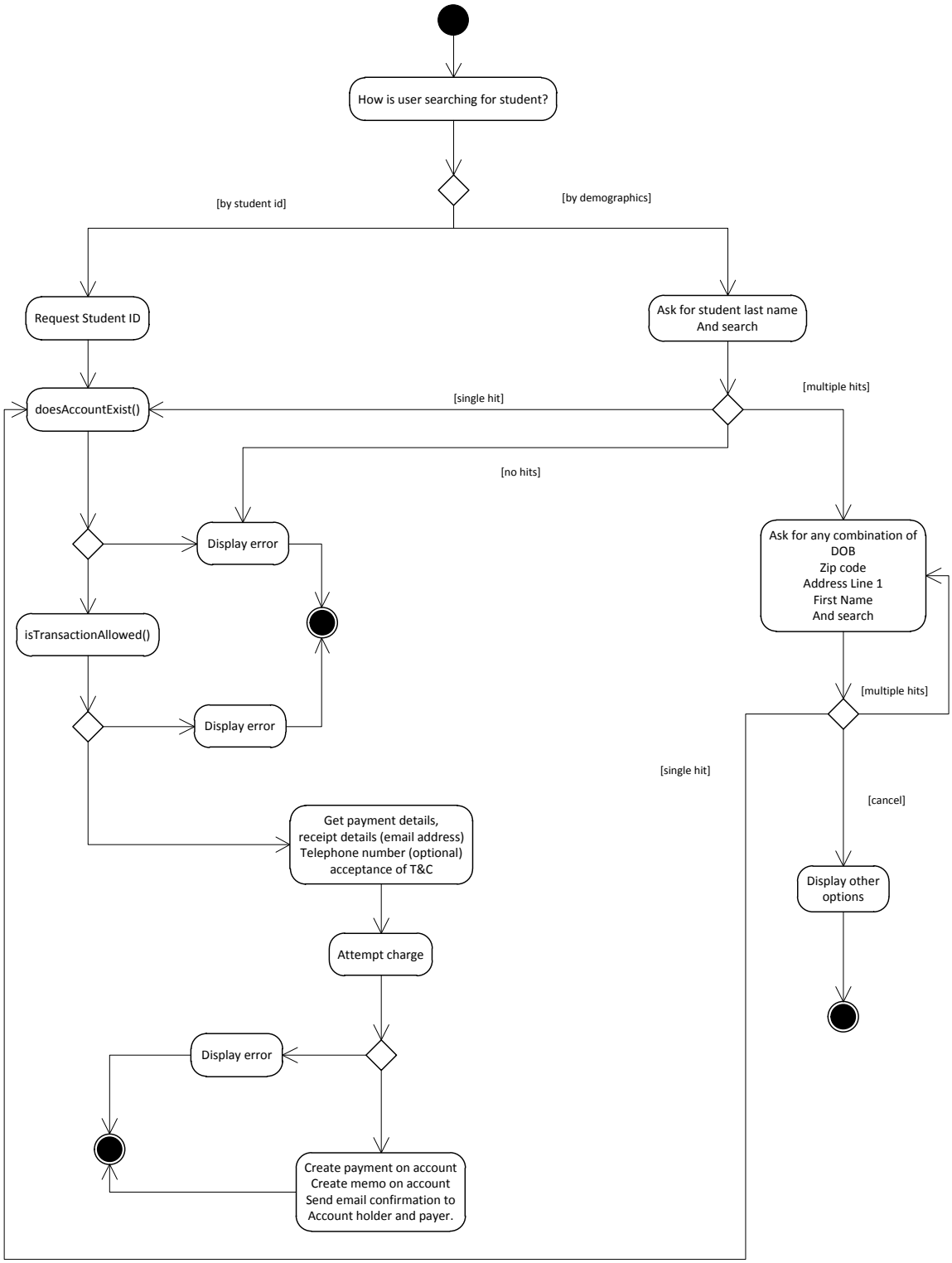
Returns Refund.


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

Miscellaneous Process Flows

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

Unauthenticated Web Portal Flow



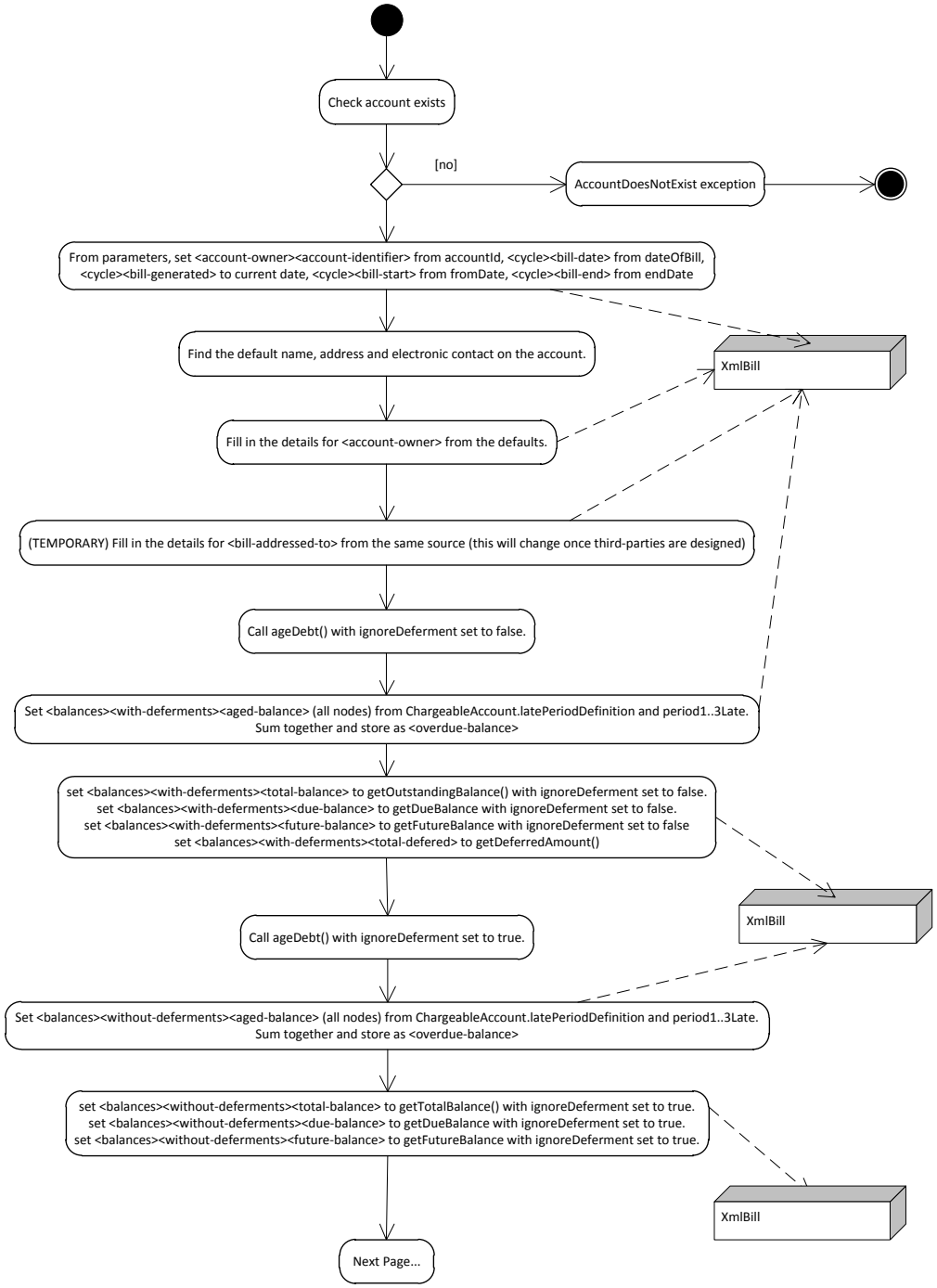
Billing Service

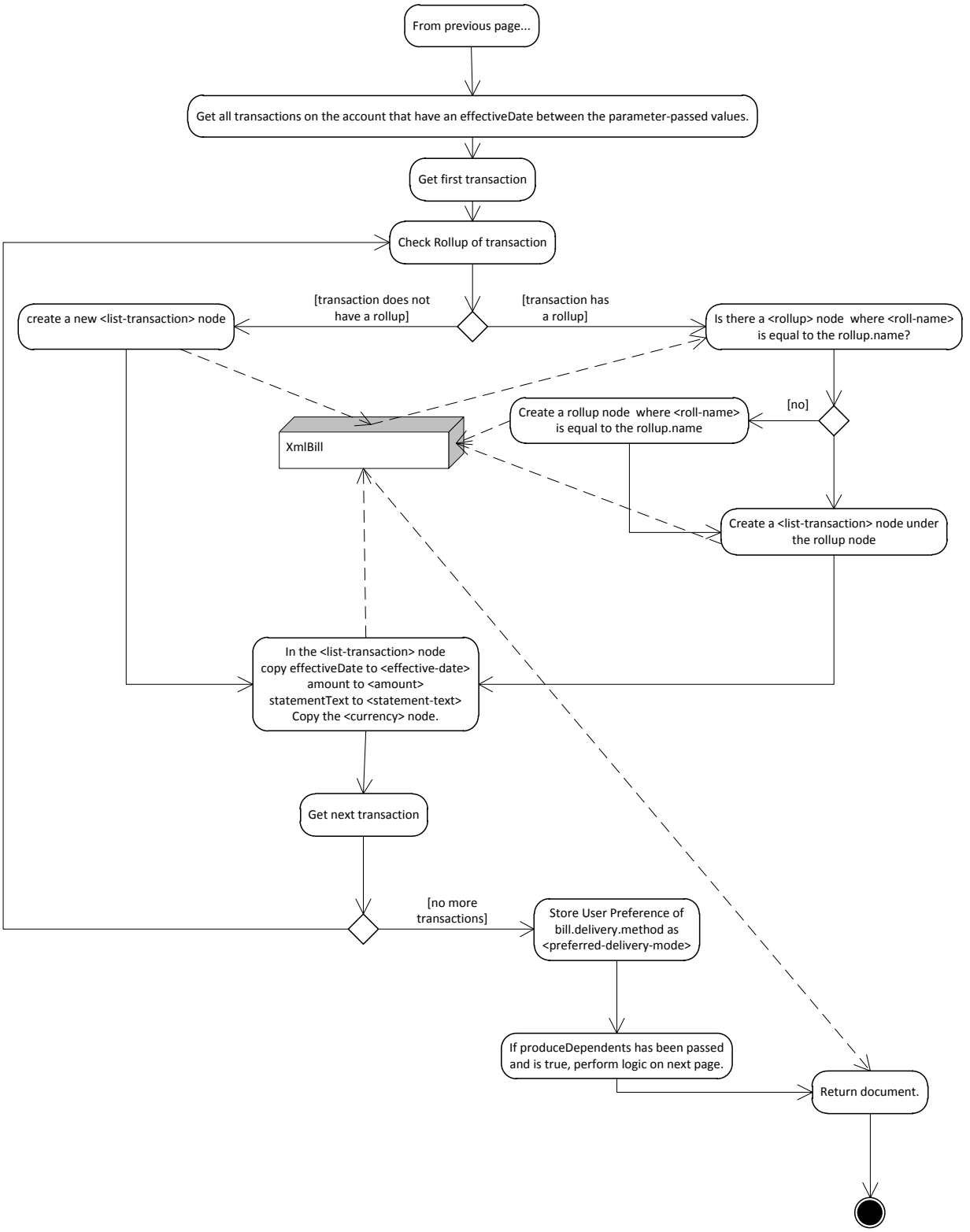
The billing service is responsible for the production of XML bills according the ksa-bill schema. These bills can be exported to other applications to produce bills in whatever format the institution prefers.

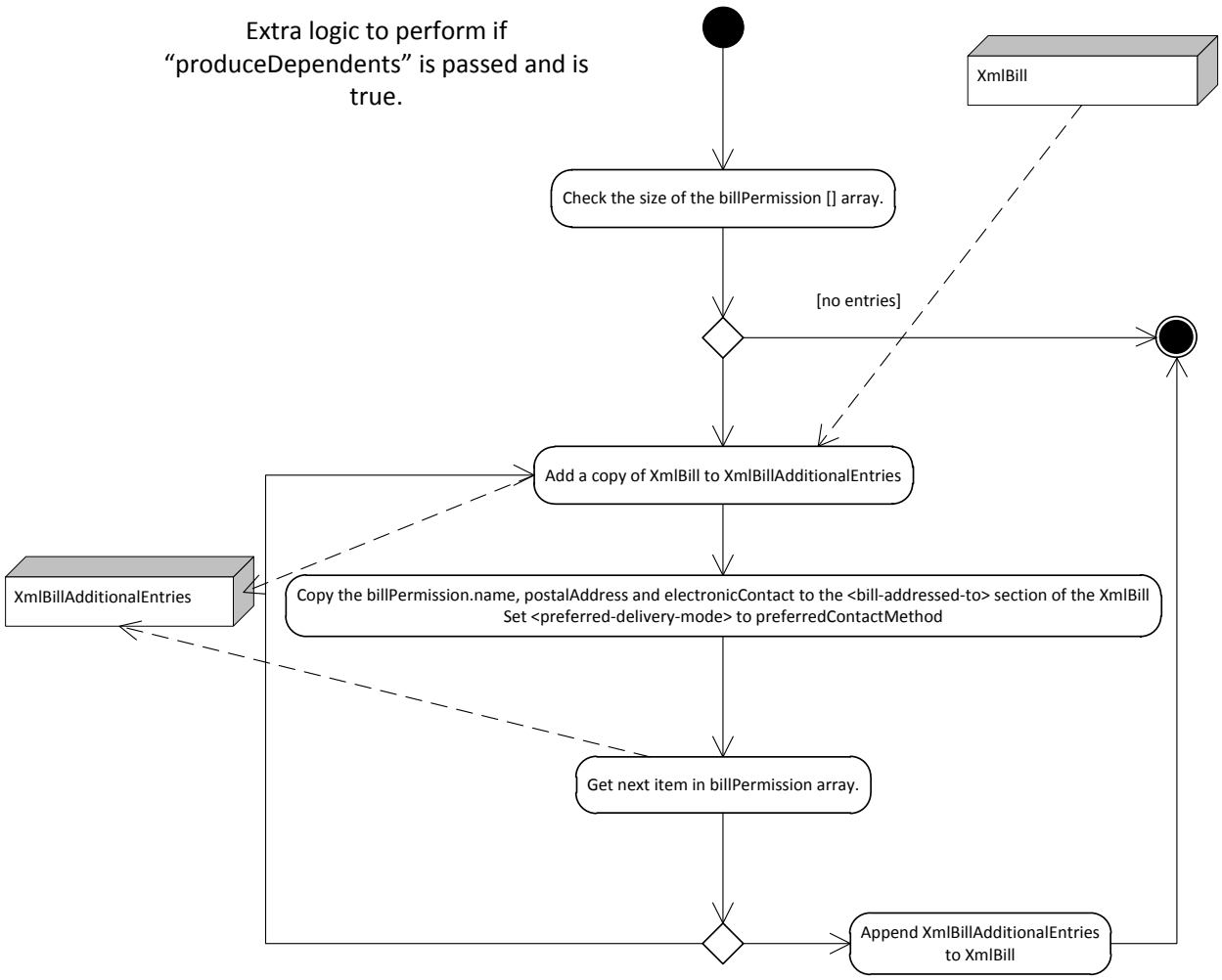
produceBill (String accountId, Date fromDate, Date toDate, Date dateOfBill, Boolean withDependents)

Returns String (XmlBill)

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
Find the default name, address and electronic contact on the account.
XmlBill
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

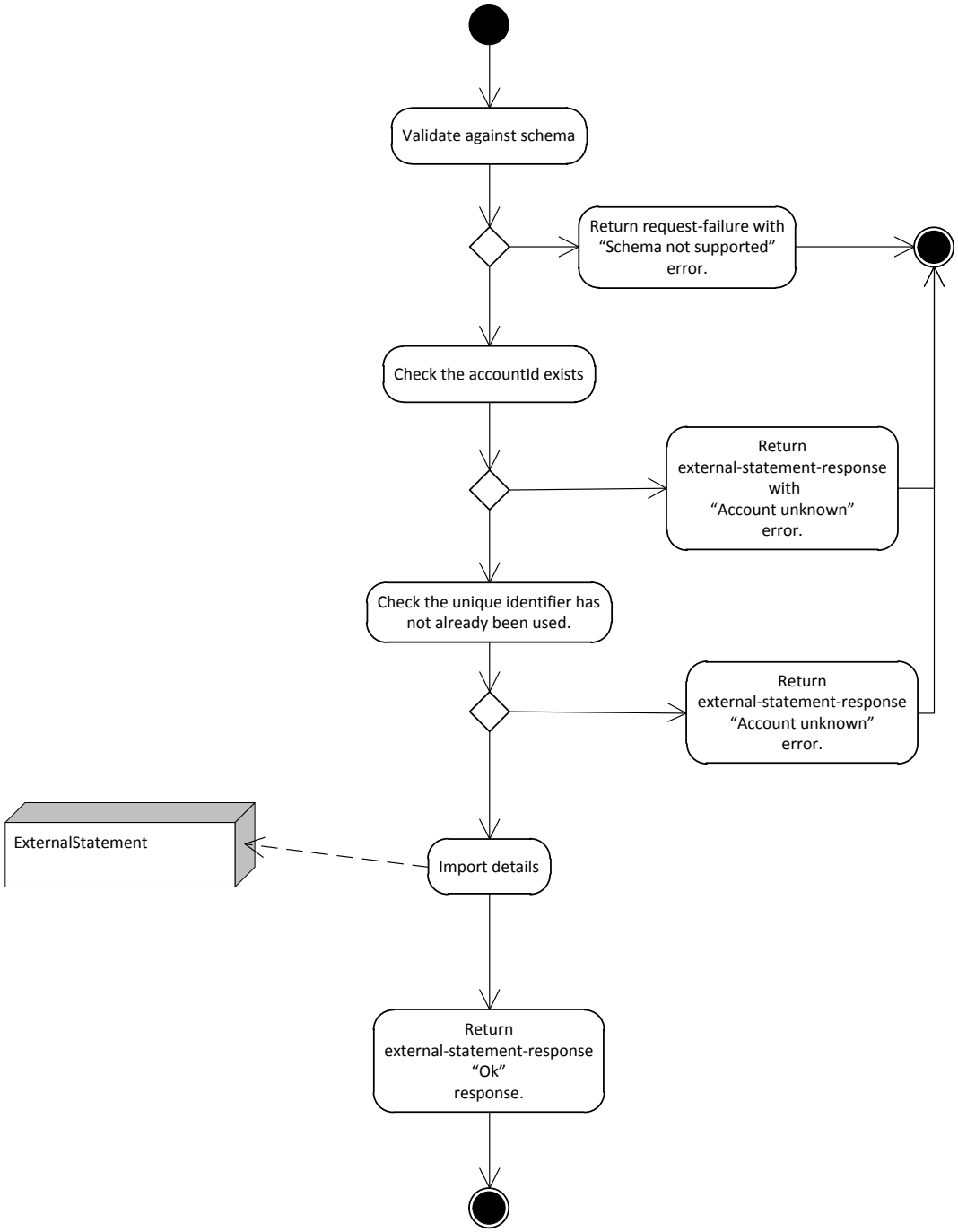
produceBill (List<String accountId>, Date fromDate, Date toDate, Date dateOfBill, Boolean withDependents)

Returns String (ksa-bill)

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

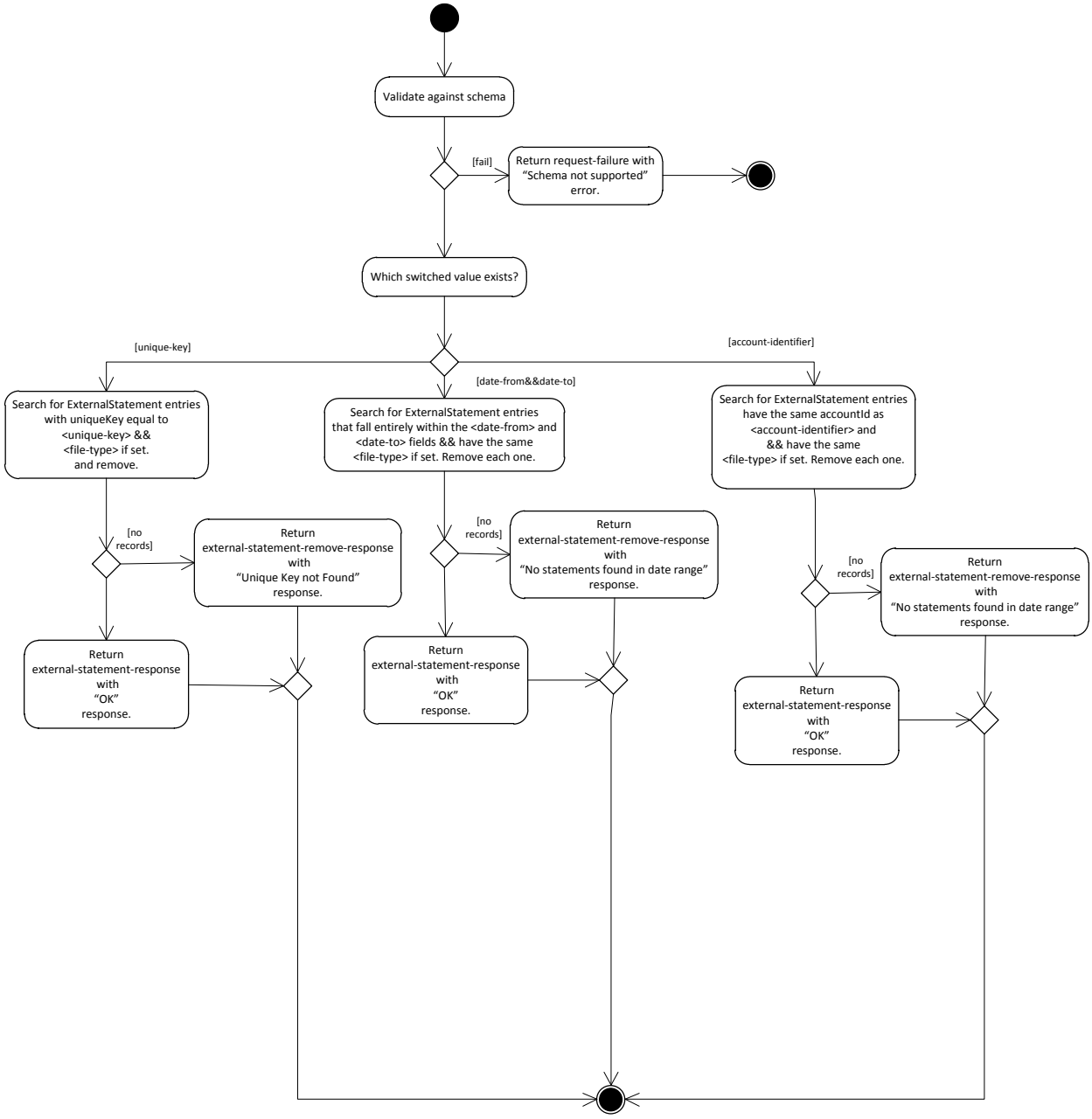
processExternalStatement(String xmlMessage)

Returns String.



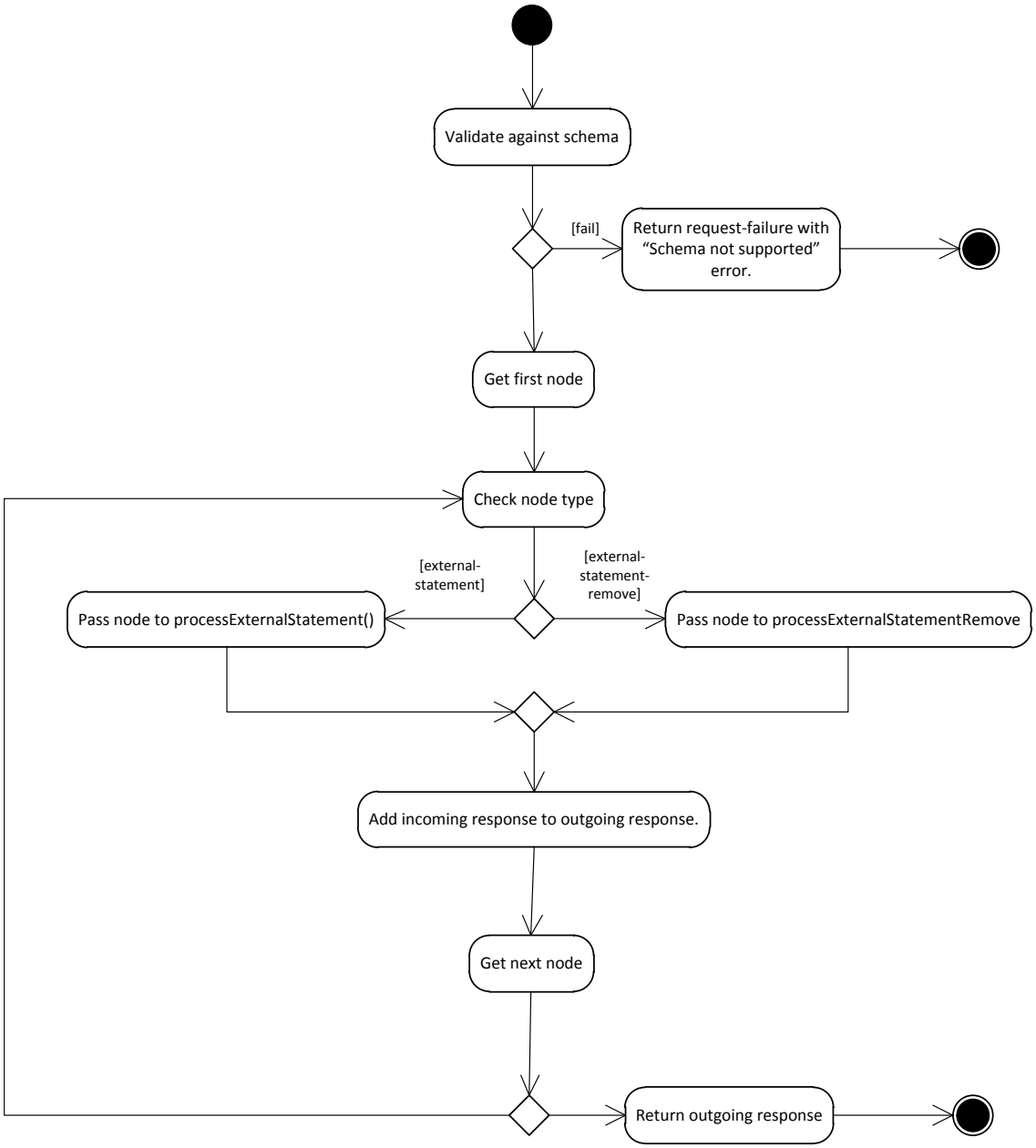
processExternalStatementRemove (String xmlMessage)

Return String.



processExternalStatementBatch (String xmlMessage)

Returns String.


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 (String accountId)

Returns void.

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, ignoring all expired deferments (isExpired = true) and pass this object to the rules engine.

paymentApplication (List<String accountId>)

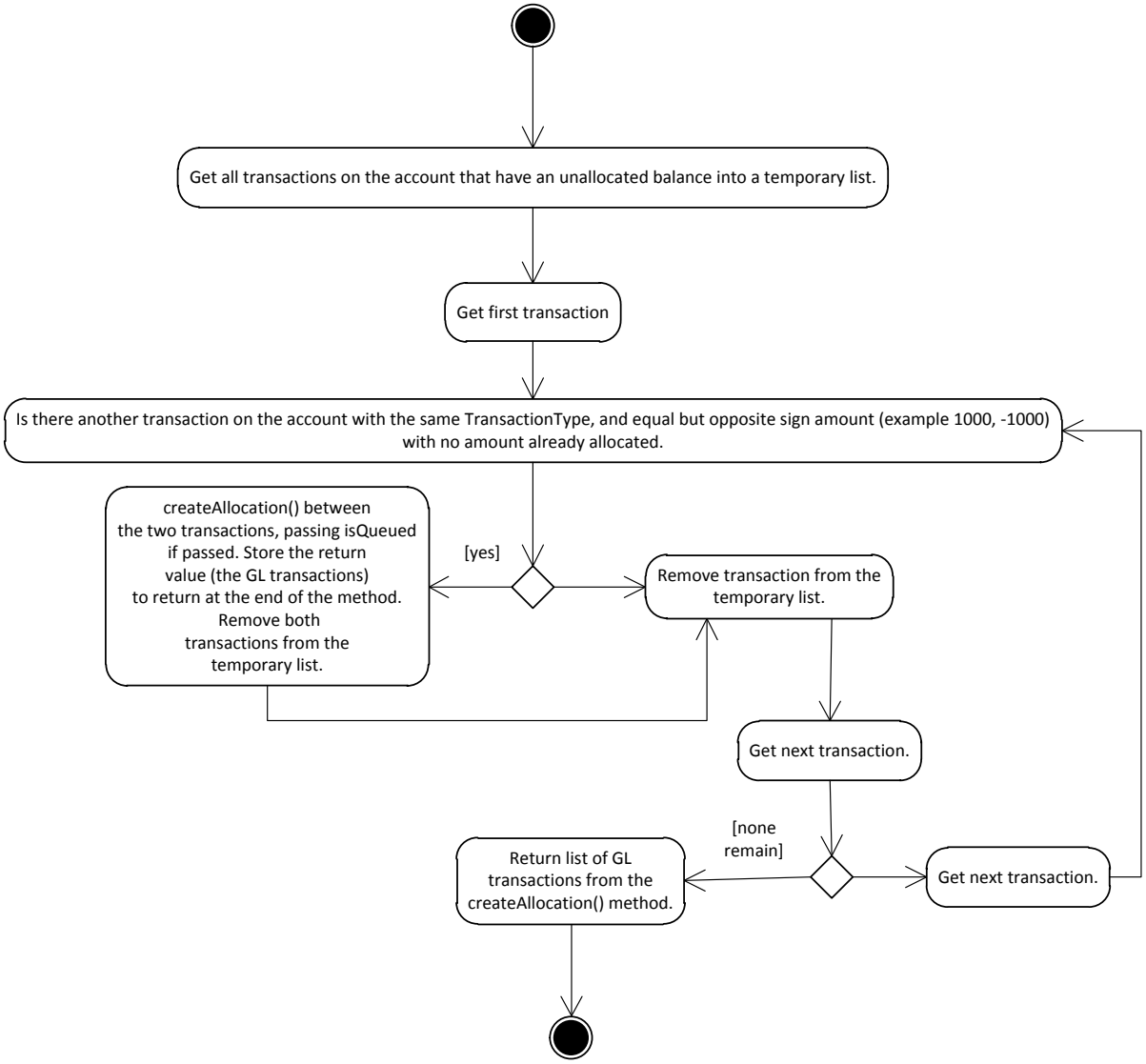
Returns void.

For each accountId, call paymentApplication()

allocateReversals (String accountId)

allocateReversals (String accountId, Boolean isQueued)

This method is used to apply “obvious” payments to their reversal. Under normal circumstances, this will not be needed, as reversals created inside of KSA will automatically be locked together. However, after an import from an external system, this allocation may not exist. This method is provided to ensure that transactions that are obviously designed to be together, are allocated together. “Obvious” means they are entirely unallocated, have the same amounts, but one is negated, and they have the same transaction type.



applyPayments (TransactionList transactionList, Boolean isQueued)

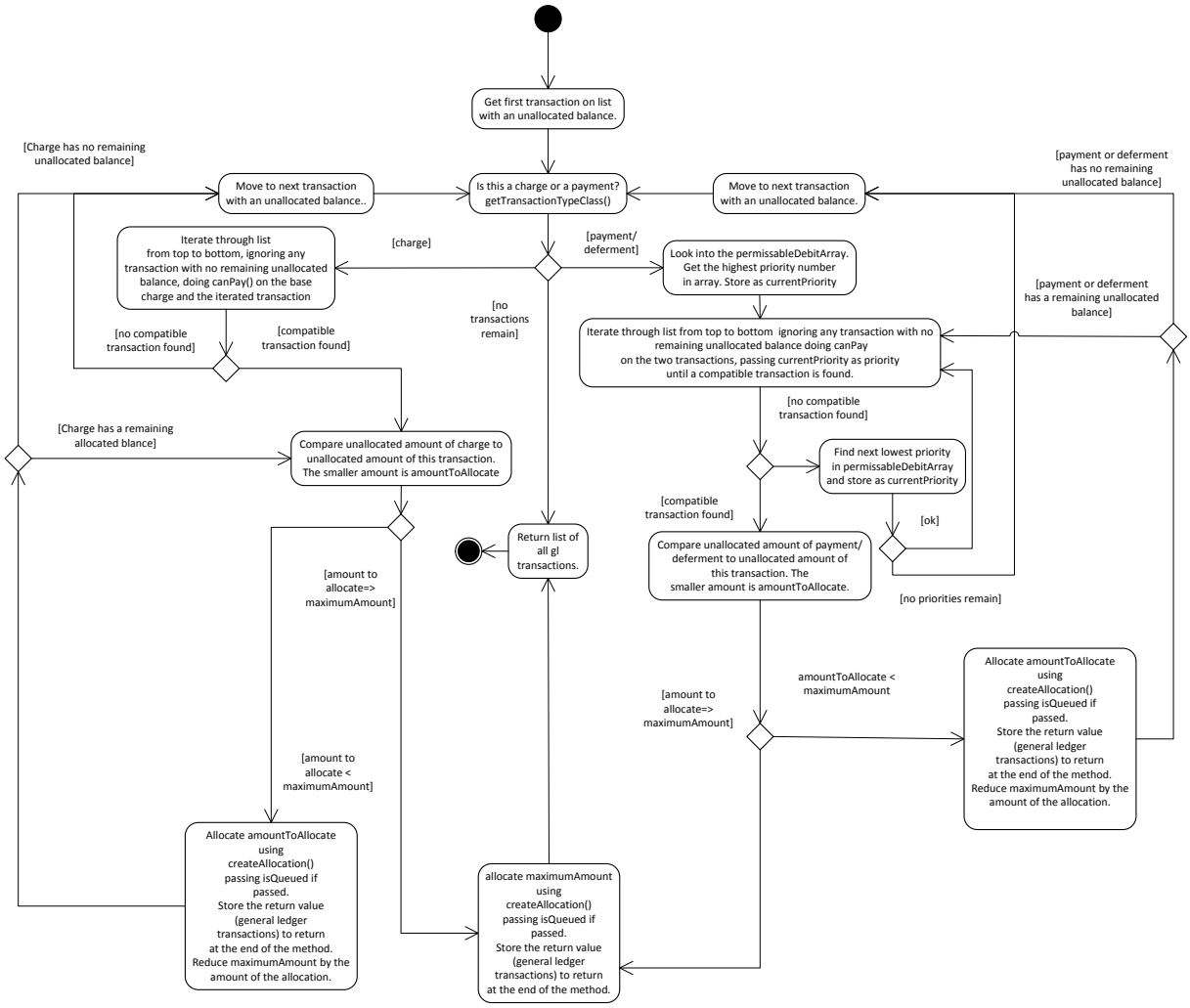
applyPayments (TransactionList transactionList, Boolean isQueued, BigDecimal maximumAmount)

Return void.

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 maximumAmount is passed, then the payment application will not allocate more than that amount.

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.



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 user 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 (TransactionList transactionList)

Returns integer.

Returns the size of the transaction[] array.

refreshList(TransactionList transactionList)

Returns TransactionList.

Remove all transactions from the list that are now fully allocated (amount = allocatedAmount + lockedAllocationAmount), or expired deferments.

getUnallocatedPaymentValue(TransactionList transactionList)

getUnallocatedChargeValue(TransactionList transactionList)

getUnallocatedDefermentValue(TransactionList transactionList)

Returns BigDecimal.

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

getRestrictedPaymentValue(TransactionList transactionList)

getUnrestrictedPaymentValue(TransactionList transactionList)

getRestrictedDefermentValue(TransactionList transactionList)

getUnrestrictedDefermentValue(TransactionList transactionList)

Returns BigDecimal.

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

calculateMatrixScore (TransactionList transactionList)

Returns Void.

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 and deferments), 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 (TransactionList transactionList, Boolean ascending)

orderByDate (TransactionList transactionList, Boolean ascending)

orderByAmount (TransactionList transactionList, Boolean ascending)

orderByUnallocatedAmount (TransactionList transactionList, Boolean ascending)

orderByMatrixScore (TransactionList transactionList, Boolean ascending)

Returns TransactionList.

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 (TransactionList transactionList)

Returns TransactionList.

Invert the list from bottom to top.

getNewList (TransactionList transactionList)

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

Returns TransactionList.

removeCharges(TransactionList transactionList)

removePayments(TransactionList transactionList)

removeDeferments(TransactionList transactionList)

Returns TransactionList.

Remove the appropriate transactions types from the list.

performUnion (TransactionList transactionList1, TransactionList transactionList2)

Returns TransactionList.

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

performIntersection (TransactionList transactionList1, TransactionList transactionList2)

Returns TransactionList.

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

performCombination (TransactionList transactionList1, TransactionList transactionList2)

Returns TransactionList.

Using the lists referenced, perform a combination between the lists and return a new list. The two lists will be merged, removing transactions on both lists. (XOR).

performSubract (TransactionList transactionList1, TransactionList transactionList2)

Returns TransactionList.

Using the lists referenced, perform a subtraction between the lists and return a new list. Any items in the second list will be removed from the first list.

filterByPriority (TransactionList transactionList, int priorityFrom, int priorityTo)

Returns TransactionList.

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 (TransactionList transactionList, Date dateFrom, Date dateTo)

Returns TransactionList.

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 (TransactionList transactionList, BigDecimal amountFrom, BigDecimal amountTo)

Returns TransactionList.

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 (TransactionList transactionList, BigDecimal amountFrom, amountTo)

Returns TransactionList.

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 (TransactionList transactionList, int matrixScoreFrom, int matrixScoreTo)

Returns TransactionList.

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).

filterByTransactionType (TransactionList transactionList, List<String transactionTypeMask>)

Returns TransactionList.

Filter out only those transactions that match the transactionTypeMask, or list of them, that is passed.

Payment Application Rules (Example)

This is an example of how the payment application rules could be applied. This example takes into account the priority of payments (which is built into the applyPayments() logic), the difficulty of applying payments, and basic financial aid methodology.

Methods that will be invoked (as example)	Description of what is being achieved.
removeAllAllocations()	Clear out the current allocation table, keeping a list of the GL entries created (the return value) Each time we apply payments, we keep a copy of these GL transactions.
allocateReversals()	Do any “obvious” allocations, where there exists transactions designed to clear other transactions. In most cases, these transactions will carry a lockedAllocationAmount, and therefore will already be allocated. Keep a list of the GL entries created.
Using getNewList() and the filterBy… methods:	Create a copy of the transaction list for each year

transactionsForYear2009.. transactionsForYear2012	that we are applying payments for (schools may elect to lock allocations at year end for prior years, to prevent allocation and reallocation of earlier years).
Using the filterBy methods, and removeCharges()/ removePayments(), allPayments allCharges allFinancialAidPayments paymentsForYear2009… paymentsForYear2012 chargesForYear2009… chargesForYear2012 financialAidForYear2009.. financialAidForYear2012	Create a list of payments, charges, and financial aid payments.
allocatePayments() using the union of chargesForYear**** and financialAidForYear****.	Allocate financial aid to charges within the year. The transaction code masking will ensure that the financial aid can only pay off the appropriate charges. Keep a list of the GL entries created.
For each financialAidForYear****, check value of payments. If it's not 0, call applyPayments() on the financialAidYear**** with all the previous year's payments, using maximumAmount of $200.	Allocate any remaining financial aid payment to prior years up to a maximum of $200, per financial aid regulations.
Refresh all the lists, and use performSubtract() to remove the remaining financialAidPayments from the allPayments list.	After this, no more financial aid payments can be allocated, so they can be removed from the payment lists.
Using performUnion, create a new list of all remaining payments and charges (with FA removed)	Now all the remaining payments and charges that can be allocated are passed.
calculateMatrixScore() on the list.	Calculate the matrix score, which dictates which charges and payment will be the hardest to allocate.
orderByMatrixScore (True)	Order this list with the lowest scores at the top. This puts the highest priority on those payments and charges that are hardest to allocate.
Call applyPayments()	Perform the payment application. Keep a list of the GL entries created.
Call summarizeGeneralLedgerTransactions() with the list of GL transactions that were recorded.	Clean up after the session, removing any duplicate entries caused by deallocation/reallocation.

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

assignToCollectionAgency (String accountId, CollectionAgency collectionAgency, String memoText)

Returns Account.

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 (String accountId, String memoText)

Returns Account.

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

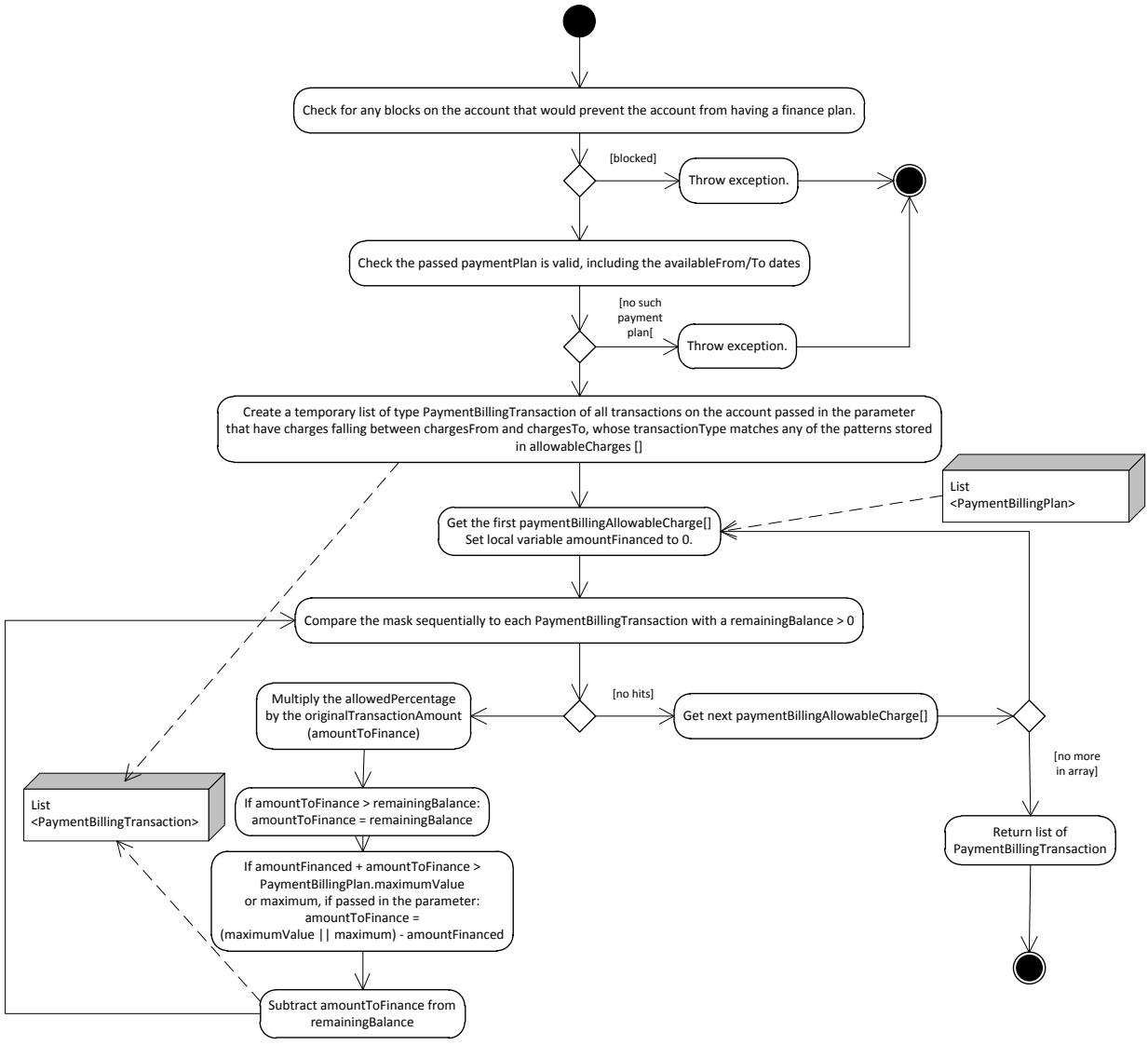
removeFromCollections (String accountId, String memoText)

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 (String accountId, PaymentBillingPlan paymentBillingPlan, BigDecimal 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.



generatePaymentBillingSchedule(paymentBillingTransaction[], paymentBillingPlan)

paymentCalculation (BigDecimal totalAmount, float percentage, Integer roundingFactor)

Returns BigDecimal.

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(String preferenceName)

Returns String.

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

setPreference (String preferenceName, String value)

Return void.

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 (String number, Integer digits)

Return String.

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”)

getUuid ()

Return String.

Returns a UUID.

Reporting Service (Part of KSA-RR)

prepare1098T (String accountId, Integer year, Integer ssnMask, Boolean noRecord)

prepare1098T (String accountId, Integer year, Integer ssnMask, Boolean noRecord, Date dateFrom, Date dateTo)

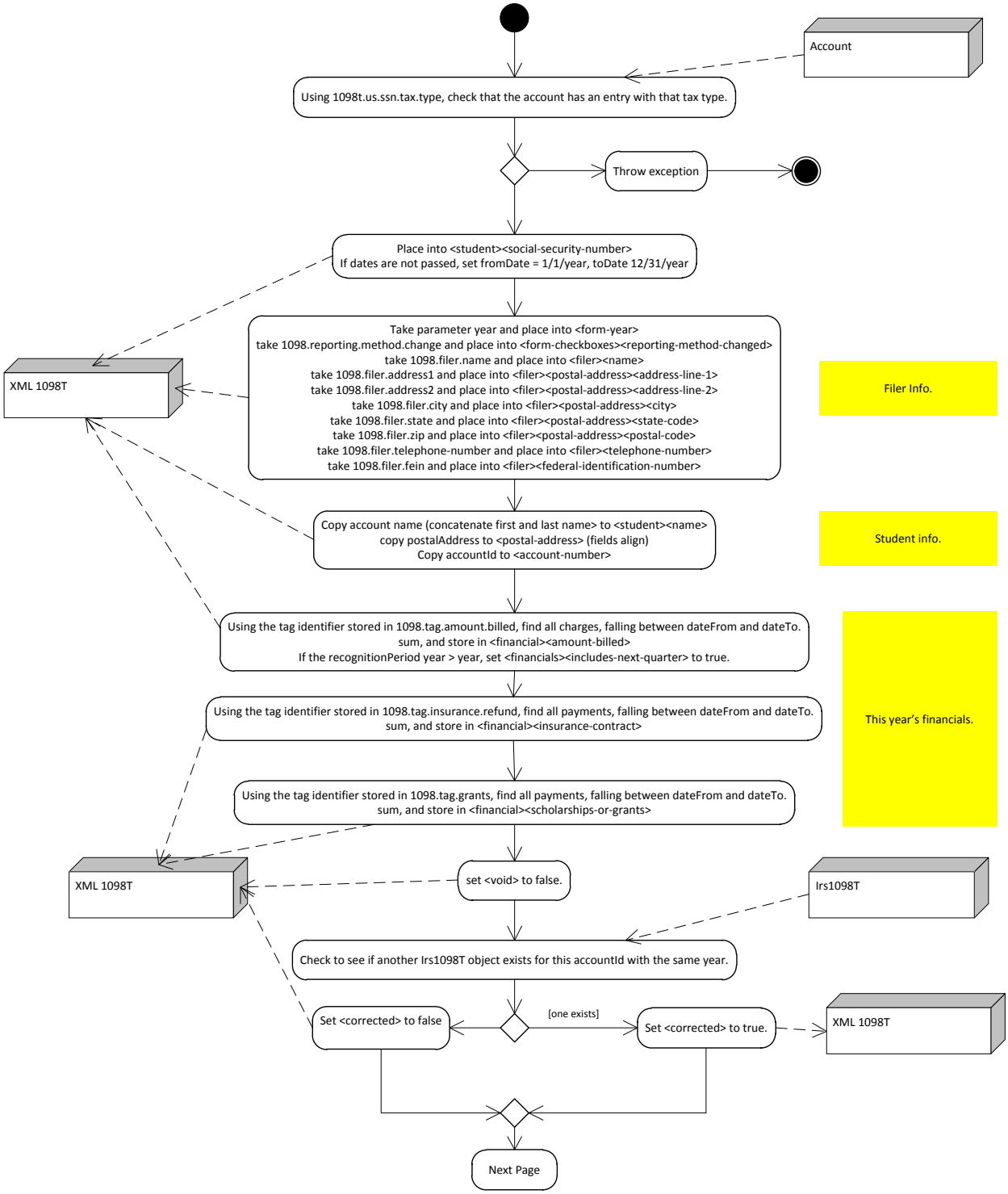
Returns String.

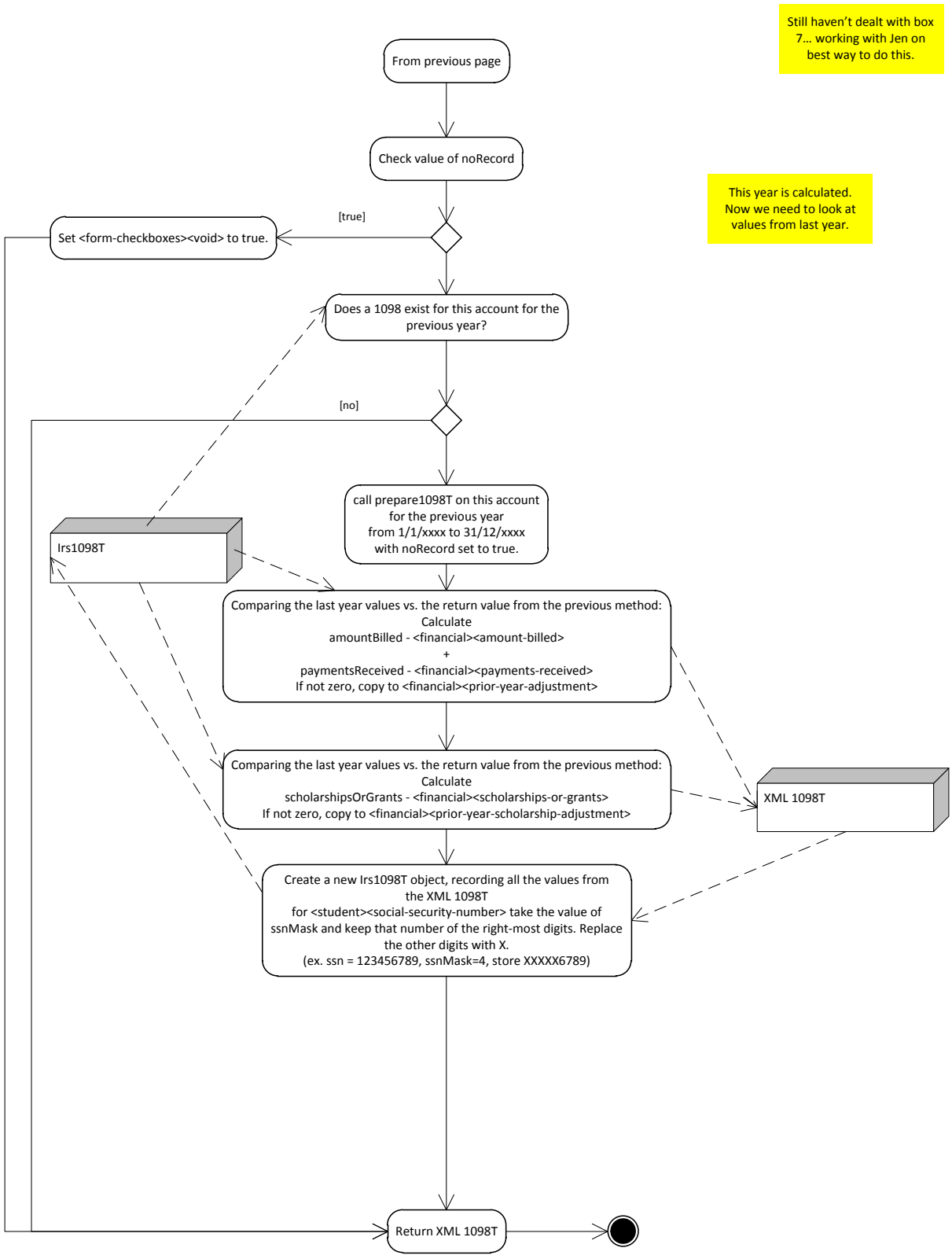
Returns an XML representation of the federal 1098T form. Note that many of the parameters for producing a correct 1098T must be established in the system preferences, and transactions must carry the appropriate tags in order to be counted correctly.

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.

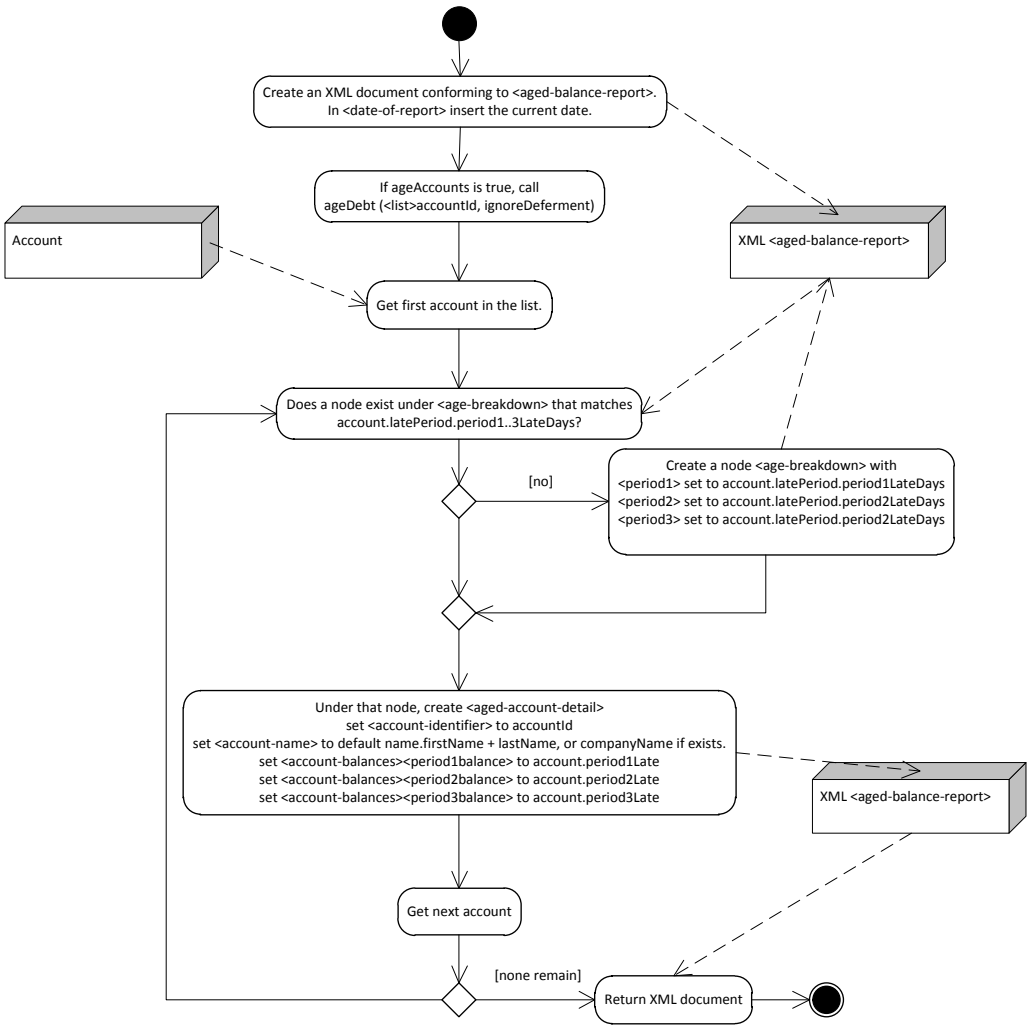

Account
Using 1098t.us.ssn.tax.type, check that the account has an entry with that tax type.
Throw exception
Place into <student><social-security-number>
If dates are not passed, set fromDate = 1/1/year, toDate 12/31/year
XML 1098T
Take parameter year and place into <form-year>
take 1098.reporting.method.change 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>
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>
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>
This year's financials.
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


Still haven't dealt with box 7… working with Jen on best way to do this.
From previous page
Check value of noRecord
This year is calculated. Now we need to look at values from last year.
[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

produceAgedBalanceReport (List<String accountId>, Boolean ignoreDeferments, Boolean ageAccounts)

Return String.

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.

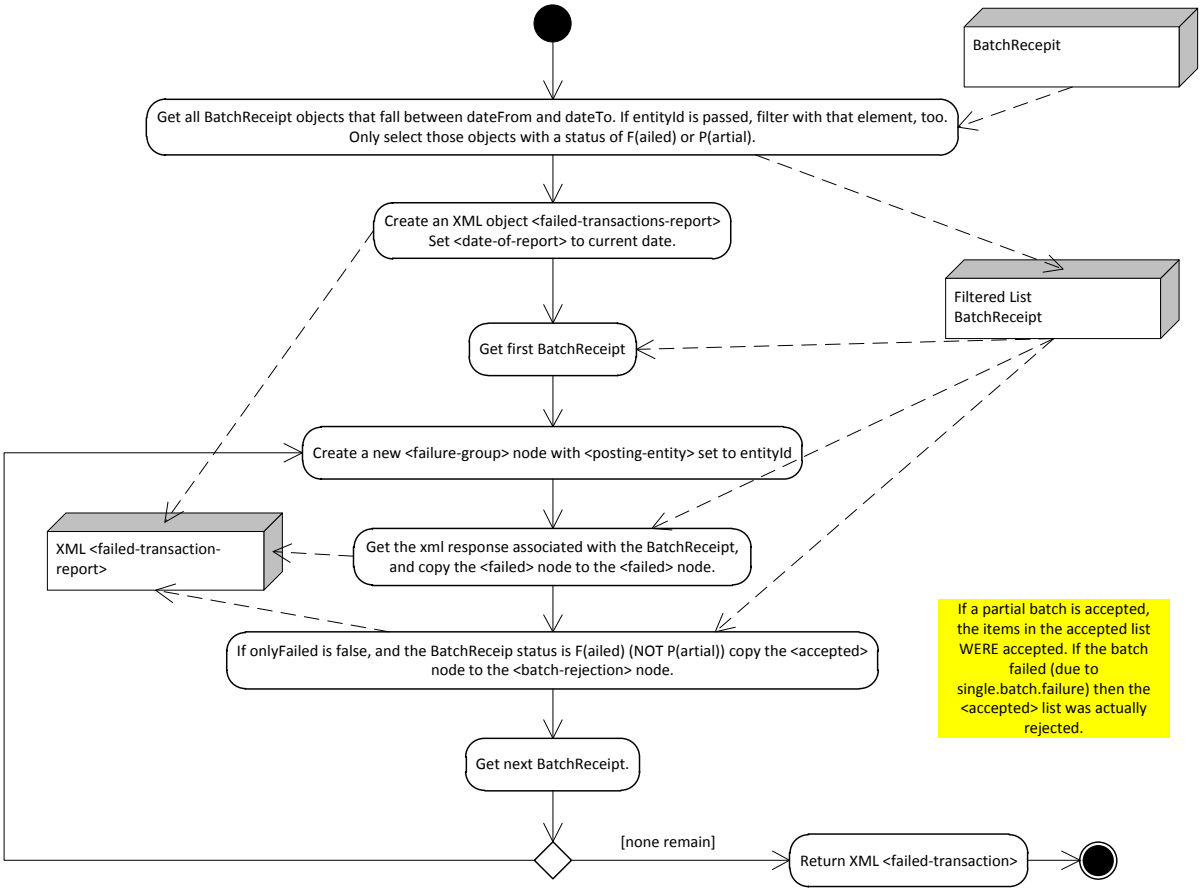


produceFailedTransactionReport (Date dateFrom, Date dateTo, Boolean onlyFailed)

produceFailedTransactionReport (Date dateFrom, Date dateTo, Boolean onlyFailed, String entityId)

Return String.

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.

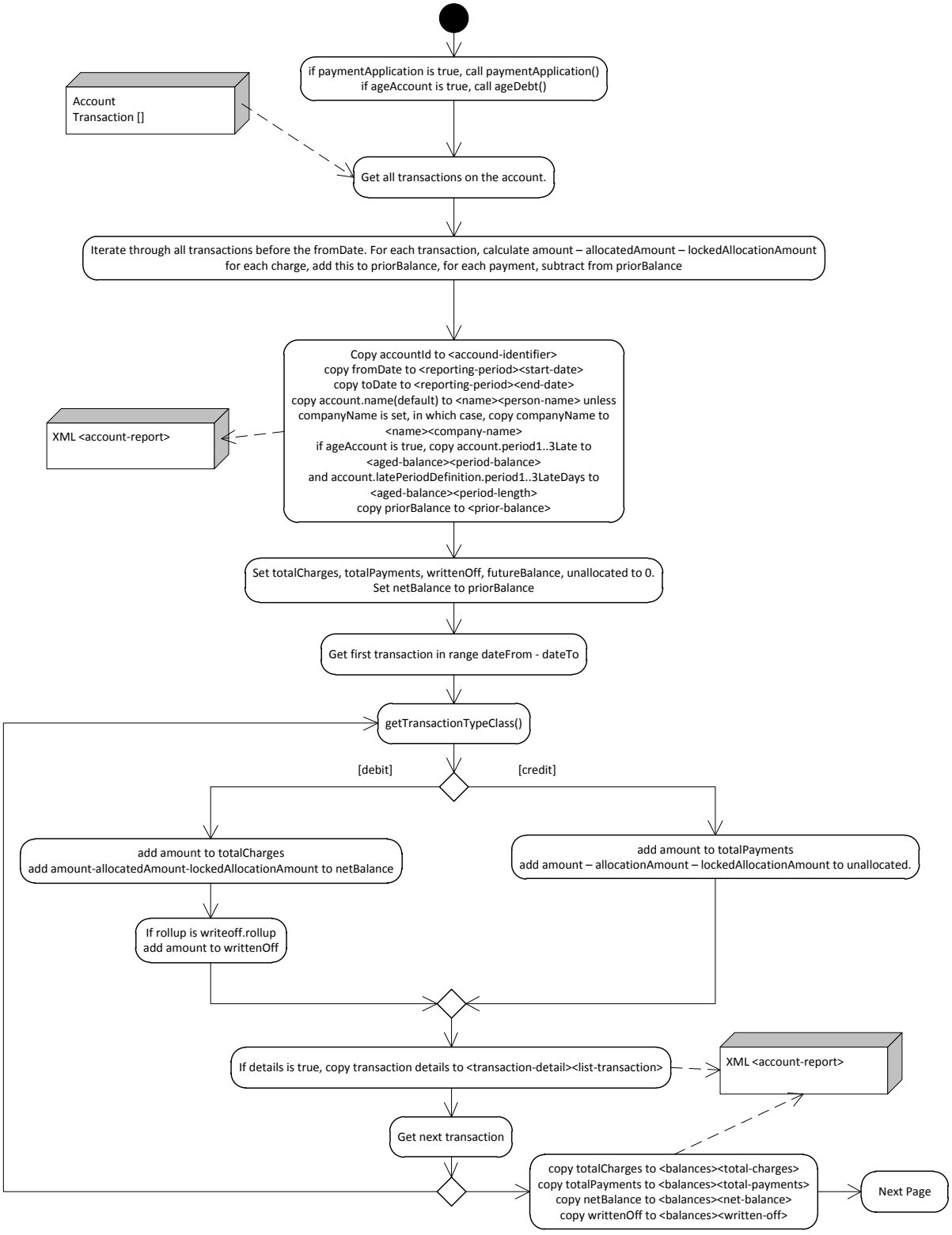


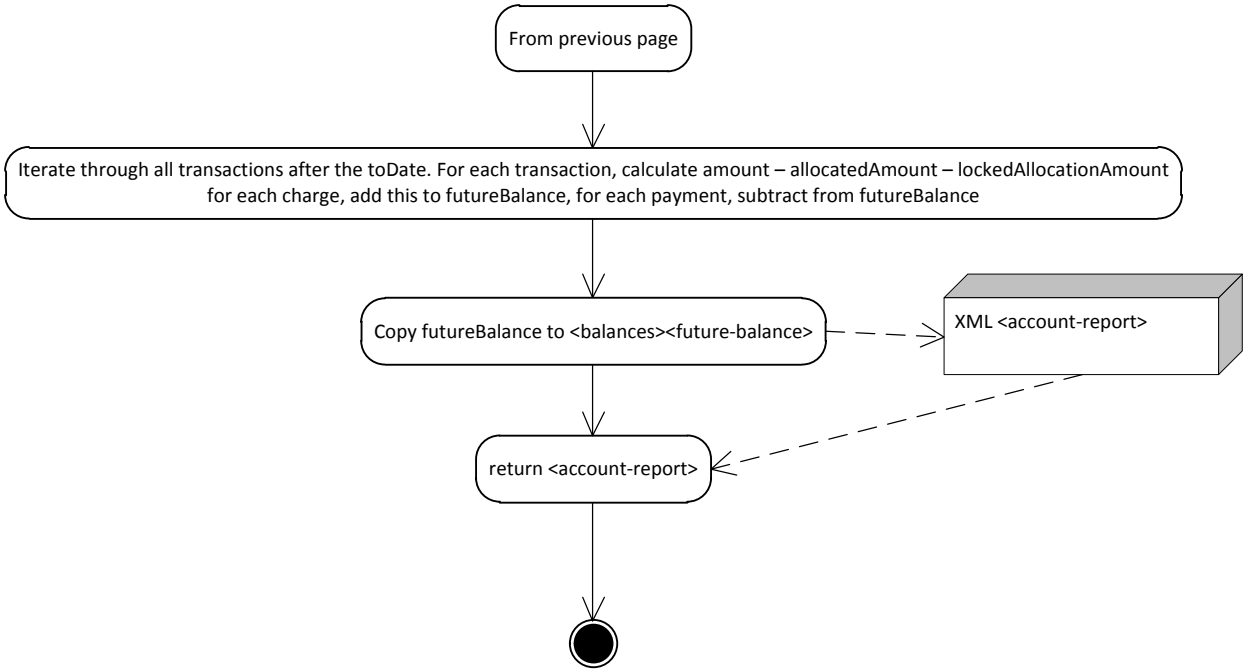
produceAccountReport (List<String accountId>, Date fromDate, Date toDate, Boolean details, Boolean paymentApplication, Boolean ageAccount)

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

produceAccountReport (String accountId, Date fromDate, Date toDate, Boolean details, Boolean paymentApplication, Boolean 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



produceReceipt (Long transasctionId)

Return String.

Kuali Student Services

The following services are Kuali services that are used by KSA. This is a non-exhaustive list.

Academic Time Period Service (ATP)

Referenced Version 1.0-rc2

(https://wiki.kuali.org/display/KULSTR/Academic+Time+Period+Service+v1.0-rc2)

ATP is used to set up time periods and is used primarily by fee management (for semester codes) and by the UI to present display options.

getAtpTypes
getAtpType

getAtpSeasonalTypes
getAtpSeasonalType
getAtpDurationTypes
getAtpDurationType

getMilestoneTypes
getMilestoneType
getMilestoneTypesForAtpType

getDateRangeTypes
getDateRangeType
getDateRangeTypesForAtpType

validateAtp
validateMilestone
validateDateRange

getAtp
getAtpsByDate
getAtpsByDates
getAtpsByAtpType

getMilestone
getMilestonesByAtp
getMilestonesByDates
getMilestonesByDatesAndType

getDateRange
getDateRangesByAtp
getDateRangesByDate

createAtp
updateAtp
deleteAtp

addMilestone
updateMilestone
removeMilestone

addDateRange
updateDateRange
removeDateRange