AccSys: AccountingSystem Schema

An object-oriented demonstration package

Bevan J Clarke

Introduction

AccSys will empower your schema to record accounting transactions and prepare accounting reports using the methods of the package, according to the principles of double-entry accounting.

AcccSys is a Jade schema ready to be imported as a package into your project. It will endow that project with a set of classes and methods. The system is delivered in two files: Dm75AccountingSystem.scm and .ddb. To make it available, make your existing schema the active schema and use the **Schema | Load** menu command and the **Merge** option to merge in these two files.

Required accounting knowledge

This is a modest package. It does not *teach* elementary accountancy—its use presupposes that the user is already adequately informed and competent in the fundamentals of double-entry bookkeeping.[1] No responsibility is accepted by the author for any use made of the package.

Other classes

The AccSys schema also has

- a class *EconAgent* which represents a class needed in the host schema to be the owner of an *AccountingSystem* and
- a form which acts as the Start-up form for the demonstration application and also as the sample output display.

General manner of use

However, do not think of AccSys as an accounting system for a human bookkeeper to use by typing in entries. Think of it instead (stripped of the two classes just mentioned) as a bookkeeping object which can be fed data (by message calls) from the owning *EconAgent* and which will return results (by message calls) to the owning system.

It is open to the implementor to enrich this interface so that, for example,

- Input might come in from real-time point of sale devices, and
- Output could be printed or otherwise displayed.

Linking the package into a host system

To link AccSys to your host application . . .

1. Define in the host a class *EconAgent* with at least three features:

name	*attribute:* a *String 50*—the public name of the business entity
balanceDate	*attribute:* a *Date,* the date of the end-point of the current accounting period (probably a year)
rAs	*reference:* an *AccountingSystem*.

   These few properties are required. All others are optional. (The *EconAgent* might be the principal owner/parent of everything in your application.)

---

[1] In fact, this is a very modest accounting package indeed. Although functional it is intended only to illustrate off-the-shelf object components and to complement the text "*An introduction to object-oriented system development, with Jade*", B. J. Clarke. For help on accounting see, for example, Clarke B. J. and A. J. Robb, *Accounting: information for action*, Pearsons Educational, Auckland NZ

2. Create an instance of *AccountingSystem*, (acs, say).

   Thereafter the *AccountingSystem* package may be referenced through rAs.

3. Execute a necessary first call on the package as follows . . .

```
ecAgent.rAs.firstCall (ecAgent, asTitle, balDate) : String ;
```

   . . . where

   - ecAgent is a string holding the proposed name of the *EconAgent*—the entity for which owns the accounting system e.g. *Global Consolidated Amalgamations Inc*.
   - asTitle is a string holding the proposed name for the accounting system (a name distinct from the name of the owning entity – e.g. this entity might have a minor accounting system called *Global Staff Social Club*).

   In this case the call would be . . .

```
ecAgent.rAs.firstCall (     "Global Consolidated Amalgamations Inc.",
                            Global Staff Social Club,
                            31-3-2002) ;
```

4. .firstCall will actually create the instance of *EconAgent*. If this does not suit the host system (perhaps because the *EconAgent* already exists) the source code of .firstCall should be altered.

The sample scripts

In the demonstration the interface form has four buttons each of which runs a demonstration: the methods test1, test2, test3 and test4.

Each of these begins with an appropriate call upon .firstCall and continues with calls upon the methods on the .cmd method of *AccountingSystem*.

All accounting is accomplished by calls upon the one method, .cmd( ) as documented below. (See method calls on .cmd( ).)

Typically each 'script' of calls . . .

- establishes a Chart of Accounts, then
- delivers a series of transactions, then
- requests one or more standard accounting statements.

The classes of an *AccountingSystem*

1. An *AccountingSystem* owns a collection of *Accounts*, which it refers to by .allAccounts.
2. An *AccountingSystem* also owns a collection of *Journals*, which it refers to by .allJournals.

   Journals are created, filled with journal entries, posted to a ledger and thereafter are unalterable. (They may only then be deleted.) Even a single journal entry must be created within a journal.

   The package always starts with one journal, "GJ, "General journal", created by .firstCall(. . .). However, any number of journals may be added. One may add more journals such as, say "Sales Journal", and one may create a series of journals spread through time, as for example:

   CR03, "Cash receipts" SJ10, “Sales journal – Sept”
   CR04, "May, Cash receipts"- BR33, “Branch consignment returns”
   CR05, "June, Cash receipts"-
   SJ08, “Sales journal – May” . . . and so on.

3. Each journal owns a collection of *JnlEntry*, referred to by .allEntries.
4. Each *JnlEntry* owns a collection of *JnlLeg*, referred to by .allLegs. (A leg is a single debit or credit within an entry.)

No journal can be posted to the ledgers until that journal balances.

5. Every instance of *AccountingSystem* owns a collection of virtual *Ledgers*, referred to by .allLedgers. Ledgers are mapped onto the single master chart of accounts as explained below. .firstCall (. . .) creates one ledger, {"GL01", "General ledger"}, and more may be added.

To create a journal or ledger the system is passed a code such as "GJ" (and a descriptive name). It appends a sequence number, e.g. "GJ31". The system will not permit a journal to acquire the same ID code as a ledger or vice versa.

Charts of accounts

The package permits, indeed requires, the user to add *Accounts* to define a Chart of Accounts appropriate to the user's needs. Accounts are added with the rAs.cmd(AccAdd, . . .) method.

Flat charts of accounts

A chart of accounts may be conventional and flat as in Figure 2.

```
Bloggs Better Boots Inc. for period ending 31 December 2000.
== Chart for GL01=General ledger 39 a/cs.
Listing elementary a/cs only.================================

Cash       Cash at Bank of Mesopotamia
AcRec      Accounts receivable
TaxDue     Taxation payable
DivPayb    Dividend payable
AcPay      Accounts payable
InvMSft    MicroSoft Inc. bonds 10%
Plant      Plant & equipment
FactBlgA   Factory building at '38 cost
DepnFact1  AcDepn on Factory building
HQBldgA    Head Office Bldg at '95 valuation
DepnHQBl   AcDepn on Head Office Bldg
CapSt      Capital @ start
Sales      Sales
IntRec     Interest income
Purch      Purchases
Salaries   Salaries
VehExp     Vehicle expenses
TaxExp     Taxation expense
RetSt      Retained income @ start
DivX       Dividend distribution
Susp       Suspense account

== End chart of accounts for ledger 'GL01'.
Last updated  1.Jan.1999, 00:00:00 ==================
```

Figure 2. A simple chart of accounts

(Account codes are simple character strings of 10-character maximum length. They may be numeric or alphanumeric.)

Hierarchical charts of accounts

Alternatively the accounts may be added so as to to build a tree structure (a hierarchy exactly like a class hierarchy.) The same ledger in principle is shown in Figure 3. Now it has added accounts, of higher level, called *structural accounts*. The accounts which have no accounts beneath them are termed *elementary accounts*, i.e. those to which transactions may be posted. All structural accounts (nodes of the tree) hold the net totals of all the accounts beneath them.

```
Bloggs Better Boots Inc. for period ending 31 December 2000.
== Chart for GL01=General ledger 39 a/cs.
Listing all a/cs.=======================================

0: R00T       Root account
1:   NetAss     Net assets
2:     Assets     Assets
3:       WCapt      Current working capital
4:         CurAss     Current  assets
5.           Cash       Cash at Bank of Mesopotamia
5.           AcRec      Accounts receivable
4:         CurLiab    Current  liabilities
5.           TaxDue     Taxation payable
5.           DivPayb    Dividend payable
5.           AcPay      Accounts payable
3:       NonCurA    Non-current assets
4:         Invest     Investments
```

```
5.           InvMSft    MicroSoft Inc. bonds 10%
4:         FixAss     Fixed assets
5.           Plant      Plant & equipment
5:           Bldgs      Buildings
6:             Facty1     Factory building BV
7.               FactBlgA   Factory building at '38 cost
7.               DepnFact1  AcDepn on Factory building
6:             HQBldg1    Head Office building BV
7.               HQBldgA    Head Office Bldg at '95 valuation
7.               DepnHQBl   AcDepn on Head Office Bldg
1:   NetOE      Net owners equity
2.     CapSt      Capital @ start
2:     RetEnd     Retained income @ end
3:       E-AT       Earnings after tax
4:         E-BT       Net surplus
5:           Rev        Revenues
6.             Sales      Sales
6.             IntRec     Interest income
5:           Exp        Expenses
6.             Purch      Purchases
6.             Salaries   Salaries
6.             VehExp     Vehicle expenses
4:         TaxExp     Taxation expense
3:       RetSt      Retained income @ start
3:       DivX       Dividend distribution
1.   Susp       Suspense account

== End chart of accounts for ledger 'GL01'.
Last updated  1.Jan.1999, 00:00:00 ======================
```

Figure 3. A hierarchical chart of accounts

When a transaction is posted to an elementary account it is immediately and automatically posted to all higher-level accounts up its branch of the tree. (Therefore the root account (named "..") always has a zero balance.)

The merits of a tree-structured Chart of Accounts such as this are that

a) it may be arranged in blocks to correspond to the terms of the 'accounting equation'.

   For example, notice that the structure of the chart in Figure 3 is . . .

   Net Assets= Net owners' equity.

   At the next level of detail we have . . .

   Working capital + Fixed assets + Investments

   = Opening capital + Retained earnings

   At the next level of detail we have, in the equity side for example, Retained earnings split into Revenue less Expenses. At a lower level of detail these in turn are broken doen into elementary revenue and expense accounts.

   In short, off the balances in this Chart of Accounts structure we can read the complete Statement of Financial Position and the Statement of Financial Performance.

b) A sub-tree corresponds exactly to the concept of a subsidiary ledger.

   For example, if we were to add new accounts beneath . . .

   ```
   5.           AcRec      Accounts receivable
   ```

   . . . they would correspond exactly to an Accounts Receivable ledger and the total of account AcRec would be the net total of all the sub-accounts.

The concept of a 'Ledger'

We said earlier . . .

5. Every instance of *AccountingSystem* owns a collection of *Ledgers*, referred to by .allLedgers. Ledgers are mapped onto the single master chart of accounts . . .

Now we see that a ledger simply points to an account which is a node in the tree of accounts. *General Ledger* points to account "..". An *Accounts Receivable* subsidiary ledger points to account AcRec. An *Expenses ledger* would point (say) to account Exp. A *Fixed asset ledger* would point to "FixAss".

This does impose a constraint in that a ledger *must* be a (sub-) tree. Therefore, if the user's

preference is to use the flat Chart structure of Figure 2 but the user wishes to group all the *Accounts Receivable* accounts into one subsidiary ledger and all *Plant* accounts into another then those two groups must be created as subaccounts of AcRec and Plant. (All accounts are always subaccounts of "..")

Classes

The principal classes within the package form the hierarchy shown in Figure 4.

Classes under the abstract class **BkObject** have common properties which support the balances and timestamps etc. that book-keeping objects need.

The miscellaneous classes under the abstract class **SupportObject** do not need those properties.

Figure 5 shows the relationships between these classes.

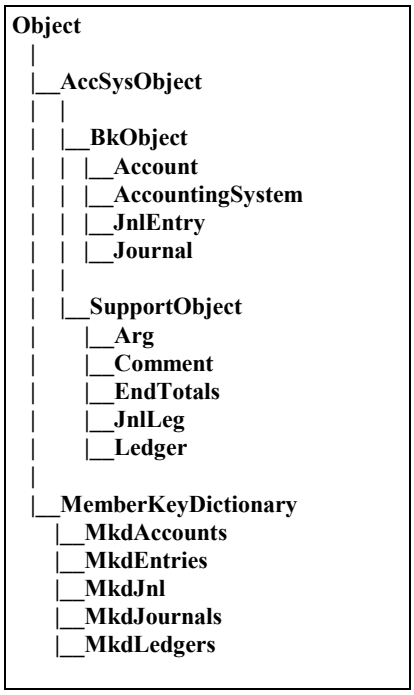


Figure 4 The accounting classes

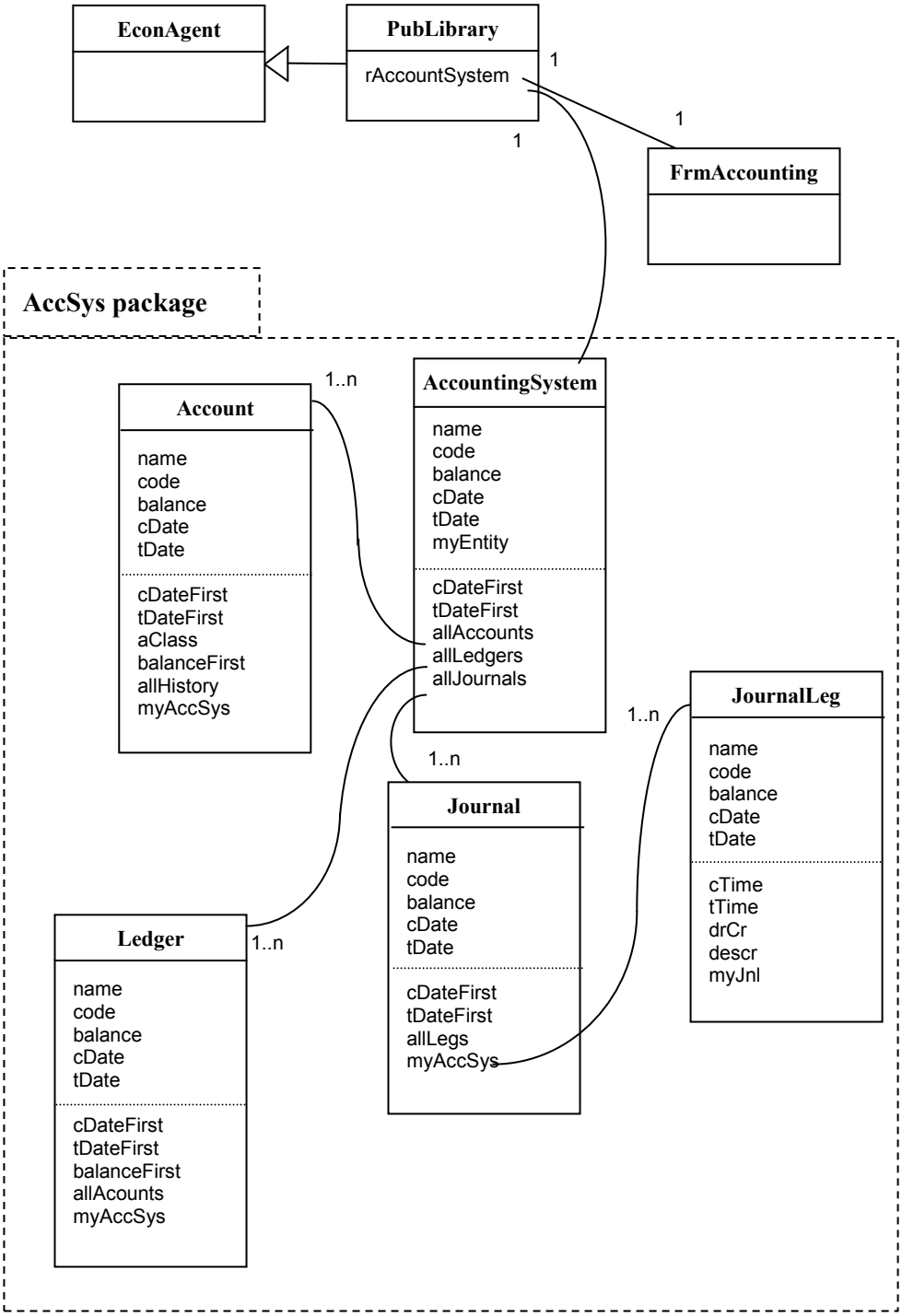


Figure 5: Major accounting classes within the accounting package

For the details of the features of the classes see the schema itself.

What are the methods that *AccountingSystem* exports?

Once the accounting system is set up and initialised with rAs.firstCall(. . .), the only method that it uses is .cmd.

The method, .cmd(. . .), takes only two arguments, <Constant> and <String>.

- <Constant> is one of the Group 1 class constants below, exported by *AccountingSystem.*
- <String> is a character-string which consists of the zero to n further arguments required by the particular sub-method. The terms are separated by ~ characters.

.cmd returns a message string beginning with "Err "if there were problems.

Constants

AccountingSystem defines the following class constants.

	Acc	*Jnl*	*Remark*	*Setting*	*Lgr*	*TrnNew*
Add	*AccAdd*	*JnlAdd*			*LgrAdd*	*TrnAdd*
DrCr						*TrnDrCr*
Dr						*TrnDr*
Cr						*TrnCr*
Force						*TrnForce*
Bal	*AccBal*	*JnlBal*			*LgrBal*	*TrnBal*
Cht					*LgrCht*	*TrnCht*
Com	*AccCom*	*JnlCom*			*LgrCom*	*TrnCom*
Del	*AccDel*	*JnlDel*			*LgrDel*	*TrnDel*
Prt	*AccPrt*	*JnlPrt*			*LgrPrt*	*TrnPrt*
Prt		*JEPrt*				
Pst		*JnlPst*			*LgrPst*	*TrnPst*
Sel	*AccSel*	*JnlSel*			*LgrSel*	*TrnSel*

In the examples which follow we show such strings and describe the arguments within the string.

For accounts

a) To add a new account to the system, AccAdd:

```
acs.cmd ( AccAdd, "~code~ location refCcode~title~drCr~Class~" ) : String ;
```

Arguments:

"~code~.....Proposed account code, e.g. Cash.

~location refCcode~ Before/After/Under refCode, where refCode is the code of an existing (i.e. already defined) account.

~title~ .....Proposed account title, e.g. Cash at Hong Kong Shanghai Bank, Shatin.

~drCr~.....D, if this account would normally be expected to have a debit balance, else C. Any other character denotes 'may be either'.

~Class~" .....One of the symbols

- A for Asset
- a for Asset Offset, e.g. Depreciation
- L for Liability
- l for Liability Offset, e.g. Provision for expected discount
- C for Capital
- R for Revenue
- r for Revenue offset, e.g. Sales returns
- E for Expense
- e for Expense offset, e.g. Purchases returns

Results: The string returned is . . .

null ..... if no error was encountered and the action was successful, otherwise . . .

"Err: Cannot parse ~ location refCcode~"

"Err: Proposed a/c '~code~ already exists within system ~system-name~"

"Err: Proposed reference a/c ~refCode~ not known! "

"Err: Op.code was ~location~. Must be 'Under' or 'After' or 'Before'."

"Err: No a/c (such as ~code~ may be added 'After' or 'Before ..'. It must be 'Under ..'."

"Err: Proposed a/c name is missing! "

Account-related results:

"Err: There are no accounts created yet.";

b) To report (return) an account balance, AccBal:

```
acs.cmd (AccBal,"~code~" ) : String ;
```

Arguments:

"~code~.....Account code. If null the 'current account' is used.

Results: The string returned is . . .

"Balance of ~code~ is ~amount~ If no error was encountered.

"Err: No account is current."

c) To append a comment to an account, AccCom:

```
acs.cmd (AccBal, "~code~" ) : String ;
```

Arguments:

"~code~.....Account code. If null the 'current account' is used.

Results: The string returned is . . .

null ..... if no error was encountered and the action was successful, otherwise . . .

"Err: There is no current account selected to add comment to."

d) To delete a closed account, AccDel:

```
acs.cmd (AccDel , "~code~ ) : String ;
```

Arguments:

"~code~.....Account code. If null the 'current account' is used.

Results: The string returned is . . .

null ..... if no error was encountered and the action was successful, otherwise . . .

"Err: Account ~code~ cannot be found to delete."

"Err: Account ~code~ may not be deleted."

e) To print (preview) an account, AccPrt:

```
acs.cmd ( AccPrt, "~code~" ) : String ;
```

Arguments:

"~code~.....Account code. If null the 'current account' is used.

Results: The string returned is . . .

null ..... if no error was encountered and the action was successful, otherwise . . .

"Err: Account code to print was blank. Using current account."

"Err: Account ~code~ specified to print, but not known. Using current account."

f) To select an account for a future command, AccSel:

```
acs.cmd ( AccPrt, "~code~") : String ;
```

Arguments:

"~code~.....Account code. If null the 'current account' is used.

Results: The string returned is . . .

null ..... if no error was encountered and the action was successful, otherwise . . .

"Err: Account ~code~ doesn't exist to select!"

For journals

a) To create a new journal, JnlAdd:

```
acs.cmd (JnlAdd, ~code~title" ) : String ;
```

Arguments:

"~code~.....Proposed two-char prefix for journal code, e.g. GJ.

~title~ .....Proposed journal title, e.g. Creditors, July.

Results: The string returned is . . .

null ..... if no error was encountered and the action was successful, otherwise . . .

"Err: ~code~ absent or too short for Journal ID code!";

"Err: ~title~ supplied for journal is blank!"

b) To report a balanced/imbalanced journal, JnlBal:

```
rAs.jnl ( JnlBal, "~code~" ) : String ;
```

Arguments:

"~code~.....Journal code, e.g. CP21. If null the 'current journal' is used.

Results: The string returned is . . .

"Balance of ~journal-name> is ~amount~If no error was encountered.

"Err: There is no current journal selected to give balance of."

c) To append a comment to a journal, JnlCom:

```
acs.cmd ( JnlCom, "~code~comment~" ) : String ;
```

Arguments:

"~code~.....Journal code, e.g. CP21. If null the 'current journal is used.

"~comment~ .....Text of the comment".

Results: The string returned is . . .

null ..... if no error was encountered and the action was successful, otherwise . . .

"Err: There is no current journal selected to add comment to."

d) To delete a posted journal, JnlDel:

```
acs.cmd ( JnlDel, “~code~” ) : String ;
```

Arguments:

“~code~.....Journal code, e.g. CP21. If null the ‘current journal is used.

Results: The string returned is . . .

null ..... if no error was encountered and the action was successful, otherwise . . .

"Err: There is no current journal to delete."

"Err: Journal, ~code~ has not yet been posted. Can't delete.";

e) To print out a journal, JnlPrt:

```
acs.cmd ( JnlPrt, “~code~” ) : String ;
```

Arguments:

“~code~.....Journal code, e.g. CP21. If null the ‘current journal is used.

Results: The string returned is . . .

null ..... if no error was encountered and the action was successful, otherwise . . .

“Err: There is no current journal selected to print";

f) To post a journal to the ledger, JnlPst:

```
acs.cmd ( JnlPst, “~code~” ) : String ;
```

Arguments:

“~code~.....Journal code, e.g. CP21. If null the ‘current journal is used.

Results: The string returned is . . .

null ..... if no error was encountered and the action was successful, otherwise . . .

"Err: There is no current journal selected to post."

“Err: Journal has already been posted! "

‘Err: Already posted: Journal ~jnlCode~ ~jnlName~ has BEEN posted on ~datePOsted~"

“Err: Unpostable: Journal ~jnlName~ out of balance by ~amount~”

“Err: Unknown a/c: Journal ~jnlName~/Tran: ~tranCode~ to account ~acct~. Posted to Susp.”

g) To select a journal for a future command, JnlSel:

```
acs.cmd ( JnlPst, “~code~” ) : String ;
```

Arguments:

“~code~.....Journal code, e.g. CP21. If null the ‘current journal is used.

Results: The string returned is . . .

null ..... if no error was encountered and the action was successful, otherwise . . .

"Err: Journal ~code~ cannot be found to select."

For ledgers

a) To add a new ledger, LgrAdd:

```
acs.cmd ( LgrAdd,    "~code~refCode~name~" ) : String ;
```

Arguments:

"~code~.....Proposed two-char prefix for ledger code, e.g. GL.

~refCcode~.....The account which is to be the node at the head of the ledger.

~name~ .....The name for the ldger.

Results: The string returned is . . .

null ..... if no error was encountered and the action was successful, otherwise . . .

"Err: ~code~ absent or too short for Ledger ID code!"

"Err: ~name~ absent for ledger!"

b) To report (return) a ledger balance, LgrBal:

```
acs.cmd (LgrBal, "~code~which~" ) : String ;
```

Arguments:

"~code~.....The code for the ledger. If null the 'current ledger' is used.

"~which~ ....."O or Opening or C or Closing or "". If null then 'C' is understood.

Results: The string returned is . . .

"Opening|Closing balance is ~amount~" if no error was encountered.

"Err: There is no current ledger.";

c) To print out a chart of accounts for a ledger, LgrCht:

```
acs.cmd (LgrCht, "~code~parameters~" ) : String ;
```

Arguments:

"~code~.....The code for the ledger. If null the 'current ledger' is used.

If the code is for a subledger then only the chart of accounts for that subledger is printed.

"~parameters~.....If the first character is "E" then only elementary accounts (lowest level accounts) are listed.

Results: The string returned is . . .

null ..... if no error was encountered and the action was successful. The chart is displayed. Otherwise . . .

"Err: No current ledger to print chart of"

d) To append a comment to a ledger, LgrCom:

```
acs.cmd (LgrCom,     "~code~comment~" ) : String ;
```

Arguments:

"~code~.....The code for the ledger. If null the 'current ledger' is used.

"~comment~ .....The commen to be attached.

Results: The string returned is . . .

null ..... if no error was encountered and the action was successful, otherwise . . .

"Err: No current ledger to to add comment to."

d) To delete a closed ledger, LgrDel:

```
acs.cmd (LgrDel , "~code~" ) : String ;
```

Arguments:

"~code~.....The code for the ledger. If null the 'current ledger' is used.

Results: The string returned is . . .

null ..... if no error was encountered and the action was successful, otherwise . . .

"Err: Ledger may not be deleted.";

e) To print (preview) a ledger, LgrPrt:

```
rAs.rLgrSys.cmd ( LgrPrt, "~code~parameters~" ) : String ;
```

Arguments:

"~code~.....The code for the ledger. If null the 'current ledger' is used.

If the code is for a subledger then only the chart of accounts for that subledger is printed.

"~parameters~ .....If the first character is "E" then only elementary accounts (lowest level accounts) are listed.

Results: The string returned is . . .

null ..... if no error was encountered and the action was successful, otherwise . . .

"Err: No current ledger to print"

f) To select a ledger for a future command, LgrSel:

```
acs.cmd ( LgrSel, "~code~" ) : String ;
```

Arguments:

"~code~.....The code for the ledger. If null the 'current ledger' is used.

Results: The string returned is . . .

null ..... if no error was encountered and the action was successful, otherwise . . .

"Err: This ledger ~code~ does not exist to be selected"

Other ledger related result messages: . . .

"Err: There are no ledgers defined yet."

For transactions

a) To add a single two-legged entry to a journal, TrnAdd

```
acs.cmd ( TrnDrCr, "~date~time~drCode~crCode~amount~description~" );
```

Arguments:

"~date~~time~ .....The date and time at which the transaction is to be recorded as having taken place-the date and time of the event in the course of the business. (The AccSys system will also automatically record the real time date and time at which the

transaction was logged by the accountant. (In the case of a real-time system (EFTPOS, say) the two would be the same.

"~drCode~.....The account to be debited.

~crCode~".....The account to be credited.

"~amount~ .....The amount of the transaction.

"~description~.....A narration. (The contra account code will be noted automatically for all two-legged transactions. For multi-legged transactions the contra account code will be "Sundries".)

Results: The string returned is . . .

null ..... if no error was encountered and the action was successful, otherwise . . .

"Err: Tran., but date is null. Date required!"

"Err. Tran. but both a/c codes are null!"

"Err: TrnAdd (or TrnDr|TrnCr) but debit|credit a/c ~drCode|crCode~ is null!" {coded to suspense)

"Err: There is no current journal selected to add transaction to."

Transaction –related results:

"Err: There are no journals to record transactions"

d) To add a credit leg to a journal, TrnCr

```
acs.cmd (TrnCr, "~date~time~ ~crCode~amount~description~" );
```

Arguments:

".....As for TrnAdd. The debit account code is merely omitted. If supplied it is ignored.

Results: The string returned is . . .

null ..... if no error was encountered and the action was successful, otherwise . . .

e) To add a debit leg to a journal, TrnDr

```
acs.cmd ( TrnDr, "~date~time~drCode~ ~amount~description~") : String ;
```

Arguments:

".....As for TrnAdd. The credit account code is merely omitted. If supplied it is ignored.

Results: The string returned is . . .

null ..... if no error was encountered and the action was successful, otherwise . . .

c) To insert a debit or credit leg, giving it an amount sufficient to balance the journal, TrnForce

```
acs.cmd ( TrnForce, "~date~time~drCode~crCode~~description~") : String ;
```

Arguments:

".....As for TrnAdd. The system determines whether a debit or a credit of 'amount' is required to balance the journal. It then uses either the debit or the credirt account code supplied. They may be the same account. If an amount is supplied it is

ignored.

Results: The string returned is . . .

null ..... if no error was encountered and the action was successful, otherwise . . .

b) To append a command to a transaction leg, TrnCom

```
acs.cmd (TrnCom, "~comment~" ) : String ;
```

Arguments:

"~comment~ .....The 'current transaction is used. This is the one last recorded.

Results: The string returned is . . .

null ..... if no error was encountered and the action was successful, otherwise . . .

"Err: There is no current journal leg selected to add comment to."

f) To delete a transaction, TrnDel

```
acs.cmd ( TrnDel, "~TransactionID~ ) : String ;
```

Arguments:

.....Impossible. A transaction may be reversed by a later transaction but it may not be expunged.

~~"~transactionID~ .....The transaction ID is a string like T.23887 which is seen attached to a transaction leg in an account or in the journal.~~

Results: The string returned is . . .

null ..... if no error was encountered and the action was successful, otherwise . . .

"Err: TrnDel not implemented yet"

For reports

a) To print a trial balance for a ledger, LgrPrt

```
acs.cmd ( LgrPrt,"~code~" ) : String ;
```

Arguments:

"~code~.....The code for the ledger. If null the 'current ledger' is used.

Results: The string returned is . . .

null ..... if no error was encountered and the action was successful, otherwise . . .

"Err: Ledger ~code~ not known."

For end-of-period

a) To close off a period, EndPer

```
acs.cmd( EndPer."~dateOld~dateNew~" ) : String ;
```

Arguments:

"~dateOld~.....The present balance date, I.e. the end of the current period.

"~dateNew~ .....The intended next balance date, I.e. the end of the period about to begin. NOTE. It is not a side effect of this command to advance the balance date of the entity whose accounting

system this is. That must be done separately since an entity might manage several accounting systems.

Results: The string returned is . . .

null ..... if no error was encountered and the action was successful, otherwise . . .

"Err: In EndPer the first date is not the present balance date. "