19. Modules

In previous chapters, we have introduced language mechanisms for describing objects and classes of a qBeta program execution. In this chapter, language mechanisms for describing the organization of programs will be introduced. A non-trivial program will usually be large, so it is desirable to be able to split such a description into smaller, more manageable units. Such units are in general called *modules*. A module is a convenient part of a program typically kept in a file (or in a database), manipulated by an editor, and translated by a compiler.

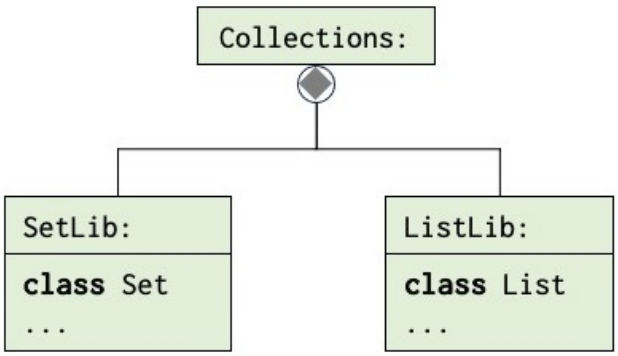
Modularization is of the utmost importance when writing programs that are more than small examples. Since the examples must be fairly small in a book like this, it is difficult to adequately illustrate and motivate modularization. We are thus using very small examples to illustrate the principles. The modularization techniques described below are absolutely necessary for large programs and highly recommended even for moderately sized programs. The reasons for this may be summarized as follows:

- Most reasonably sized programs can be conveniently split into modules of logically related elements, since it is cumbersome to handle large programs. Large programs are easier to understand if split into a number of smaller units.
- When editing it is easier to manage a number of small modules instead of one large program.
- When several people are working on a project, each person can work on his/her own set of modules.
- Modules can be saved in a library and shared by several programs. Good modularization mechanisms will thus improve reusability of code as well as of designs.
- A module must have a well-defined interface as discussed in . As discussed in , this makes it possible to prevent users of a module from seeing details about data representation and implementation of algorithms.
- Certain modules may exist in several variants. One example of this is different implementations of the same (interface) module. Another example is variants of a module corresponding to different computers. If a module has several variants it is important that the common part of two or more variants exists in only one copy. It should only be necessary to separate out the code that differs between variants, otherwise maintenance becomes complicated, since the same change may have to be made for several variants. In this version of the book, we do not discuss language mechanisms for supporting variants.
- A module may be used as a unit to be separately compiled. When developing a program, it is in most cases a good idea to be able to try out even small changes in the program including if a given change may compile and works as expected. For this to work, the compiler must be fast enough to avoid breaks when waiting for the compiler. For small programs this is usually not a problem, but for very large programs, the compilation time may be long. When changing parts of a large program, it is not acceptable to be forced to recompile the whole program, since this may take several hours. For this reason, some language implementations make it possible to separately compile modules such that they do not have to be recompiled for changes not affecting these modules. With separate compilation of modules, only the modules that have been changed and those that are affected by these changes have to be recompiled. Below we explain how one module may be affected by changes in another. Separate compilation is not supported by the current implementation of qBeta, but compilation time should not be a problem for the program examples used in this book.

In qBeta, a *module* is a description of a singular object. A singular object generated from a module is called a *module object*. Modules are organized in a hierarchy of nested intrinsic module objects that follow the general mechanism of nesting and intrinsic objects in qBeta. A module may also be considered to be a program – we return to this below.

As an example, all collection classes are defined within a module called `Collections` and each kind/type of collection is defined as a nested module in `Collections`:

```
Collections: obj
   SetLib: obj
      class Set:
         ...
   ListLib: obj
      class List:
         ...
```



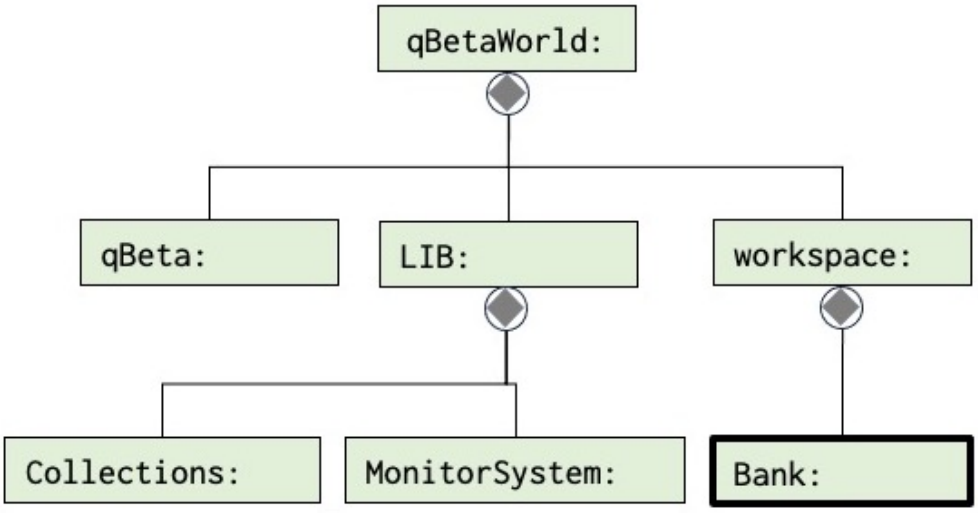
As mentioned, the module hierarchy of qBeta is basically a (large) nested structure of modules. The top-level module is called `qBetaWorld`. There are a number of modules nested within `qBetaWorld`:

- The module `qBeta` contains all basic declarations of qBeta.
- The `LIB` module contains all standard library modules for `qBeta` including `Collections`.
- The `MonitorSystem` from section is another example of a module.
- The `Workspace` module is a a place for a user of qBeta to place his/her example modules. One example may a module containing the `Bank` example used in previous sections.

The following example shows part of the `qBetaWorld` modules:

```
qBetaWorld: obj
   qBeta: obj
      ...
   LIB: obj
      Collections: obj
         ...
```

```
      MonitorSystemLib: obj
            ...
   Workspace: obj
      Bank: obj
         class Account:
            -"-
         class Customer:
            -"-
         -"-
```



The module hierarchy is mapped to a corresponding folder hierarchy and each folder has a source file containing the corresponding qBeta code. There will thus be a folder name `qBetaWorld`, which contains folders, `qBeta`, `LIB` and `Workspace`. The folder `LIB` contains folders `Collections`, and `MonitorSystemLib`, etc. The folder `SetLib` has a file `SetLib` that contains the code for class `Set`, etc.

Note that in the diagram above we have showed the module hierarchy both as a composition diagram, with objects containing intrinsic objects, and a nesting hierarchy since the modules are descriptions of singular part objects that are nested in each other. The grey rhomb illustrates composition and the circle illustrates nesting – see also .

The advantage of organizing modules in a nested hierarchy of singular object descriptors is that visibility of modules and their attributes follows the ordinary scope rules of qBeta. The module `Bank` may e.g. access attributes in the `Collections` module. The `Bank` object has a data-item `theAccountFile`, which is of type `Set(Account)` where `Set` is defined in `Collections`.

Since modules are just nested object descriptors, all modules are visible from any module at any place in the hierarchy, but constrained by access modifiers like `%public`, `%private`, `%protected`, and `%package`.

As mentioned, the `Bank` may be organised as a module. As shown in chapter , the `Bank` uses class `Set` to contain the accounts of the `Bank`. In the next example, we show how class `Set` may be accessed by means of an attribute accessor for objects.

```
Bank: obj
   class Account:
      -"-
   -"-
   theAccountsFile: obj LIB.Collection.SetLib.Set(Account)
```

As can be seen, the declaration of `theAccountsFile` makes use of a full path `LIB.Collections.SetLib.Set` to class `Set`. If one finds this to be too verbose, it is possible to make all public declarations in a given module visible using the `%visible` property:

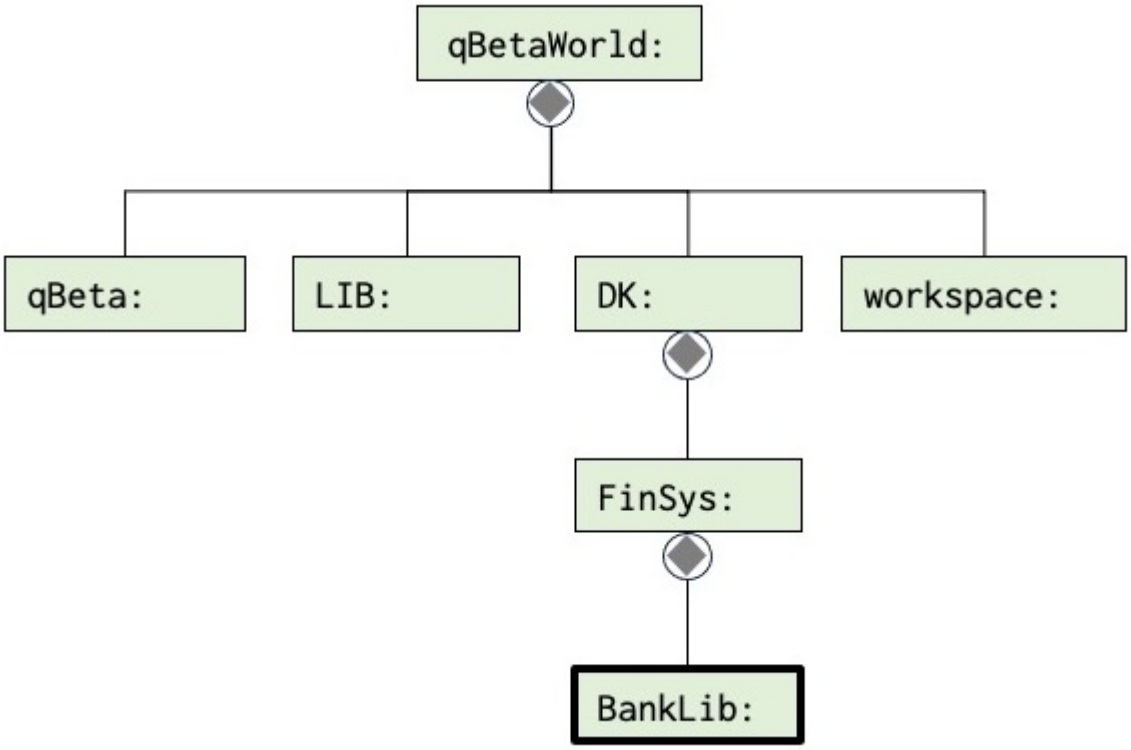
```
Bank: obj
   %visible LIB.Collections.SetLib
   class Account:
      -"-
   -"-
   theAccountsFile: obj Set(Account)
```

Organizing Bank as a general module

At some point the code for `Bank` may have reached a stage where you will make it available for other programmers as a library. You then have to place it in a module other than `Workspace`. For this example, we place it in a module called `BankLib`.

Next we have to decide where to place `BankLib` in the module hierarchy. Other programmers may use the name `BankLib` for other purposes in which case there may be name conflicts. To avoid name conflicts it is in general recommended to place a module in a module that has a name that is the name of the reversed Internet domain of the organization that has developed the code. If we assume that `BankLib` is developed by a Danish company called `FinSys` (Financial Systems Ltd.), then we place `BankLib` in the module `DK.FinSys`– assuming that the Internet domain of the company is `FinSys.DK`. `BankLib` may thus be referred to as `DK.FinSys.BankLib`:

The convention for using reverse Internet domain names was original proposed for package names in Java. Modules as presented here has some resemblance of packages – se XXX



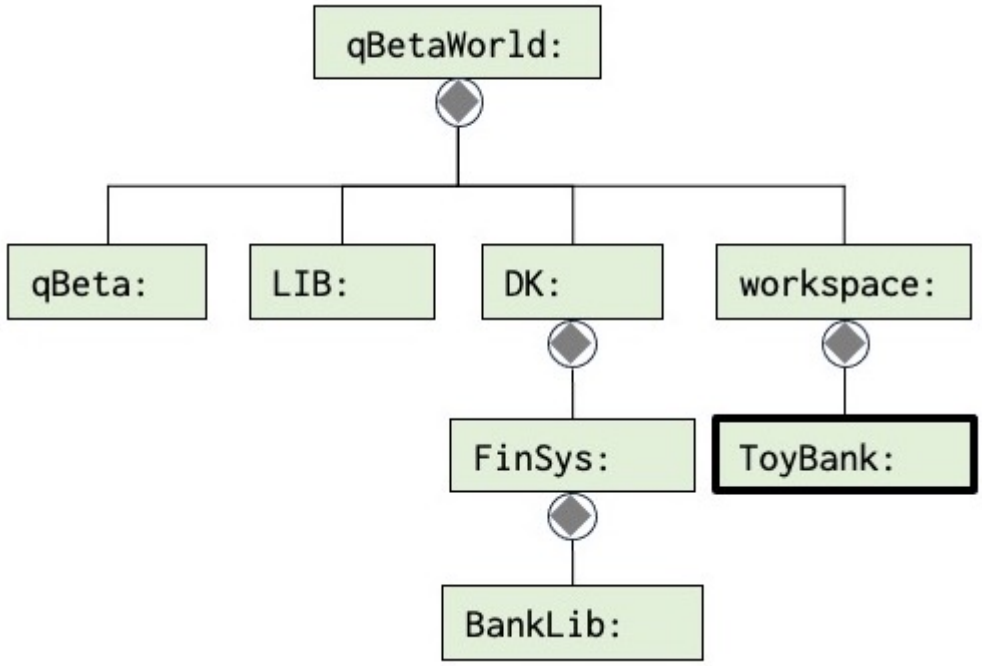

```
BankLib: obj
  class Bank:
      theAccountsFile: obj Set(Account)
      ...
  class Account(owner: ref Customer):
      -"-
  class Customer:
        -"-
```

As can be seen, we have placed class `Account` and class `Customer` in the `BankLib` module. In addition, we have defined `Bank` as a class and not as an object as in the example above. The reason is that different programmers should be able to create their own `Bank` objects. In addition the classes `Account` and `Customer` are not defined within `Bank`, but in `BankLib`, and they are used in `Bank`, e.g. `Account` in in the declaration of the `theAccountsfile`, and `Customer` in the definition of class `Account`.

We only show the program text for (part of) `BankLib`. But even if modules like `BankLib` are placed in separate folders and files, it exists in the context for the whole of `qBetaLib` and the semantics is as if the whole library is one (large) text as shown in the first examples above.

A user may then create his own bank using the `BankLib` module and start by placing it in `Workspace`:

```
ToyBank: obj DK.FinSys.BankLib.Bank
   aClerk: obj
       handle:
          -"-
   aClerk.handle
```



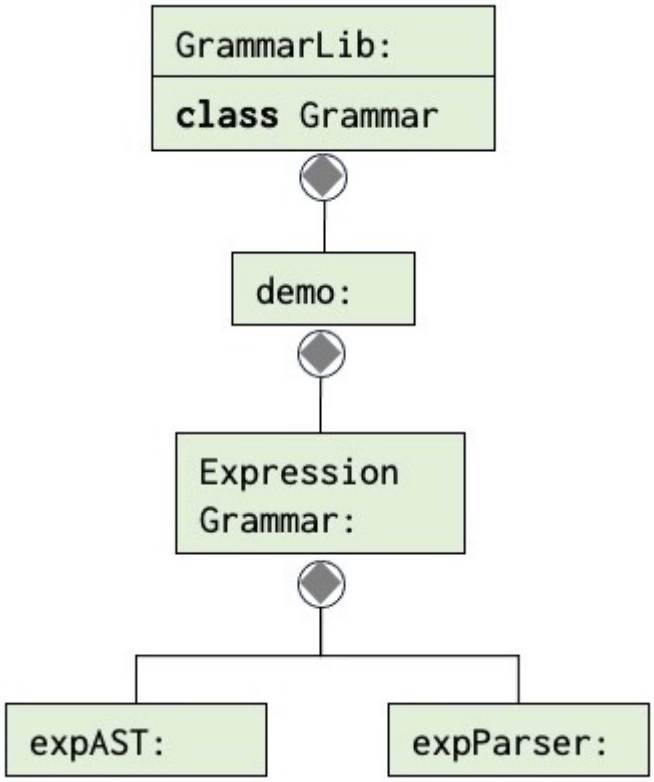
Here we have just made a simple `ToyBank` that contains the object `aClerk` from section and makes use of the module `DK.FinSys.BankLib`. Note that `ToyBank` is derived from `DK.FinSys.BankLib.Bank`.

The `ToyBank` object is a module but it is also a program. In general a program is a module. In principle any module could be viewed as a program, but for a module to be considered a program it should represent a system in a given domain and/or solves a specific problem. However, there is no clear definition of when a module should be considered a program since any module can be executed.

Organizing a program as multiple modules

A program may also be split into modules and here we show how the example of an expression grammar from section may be organized.

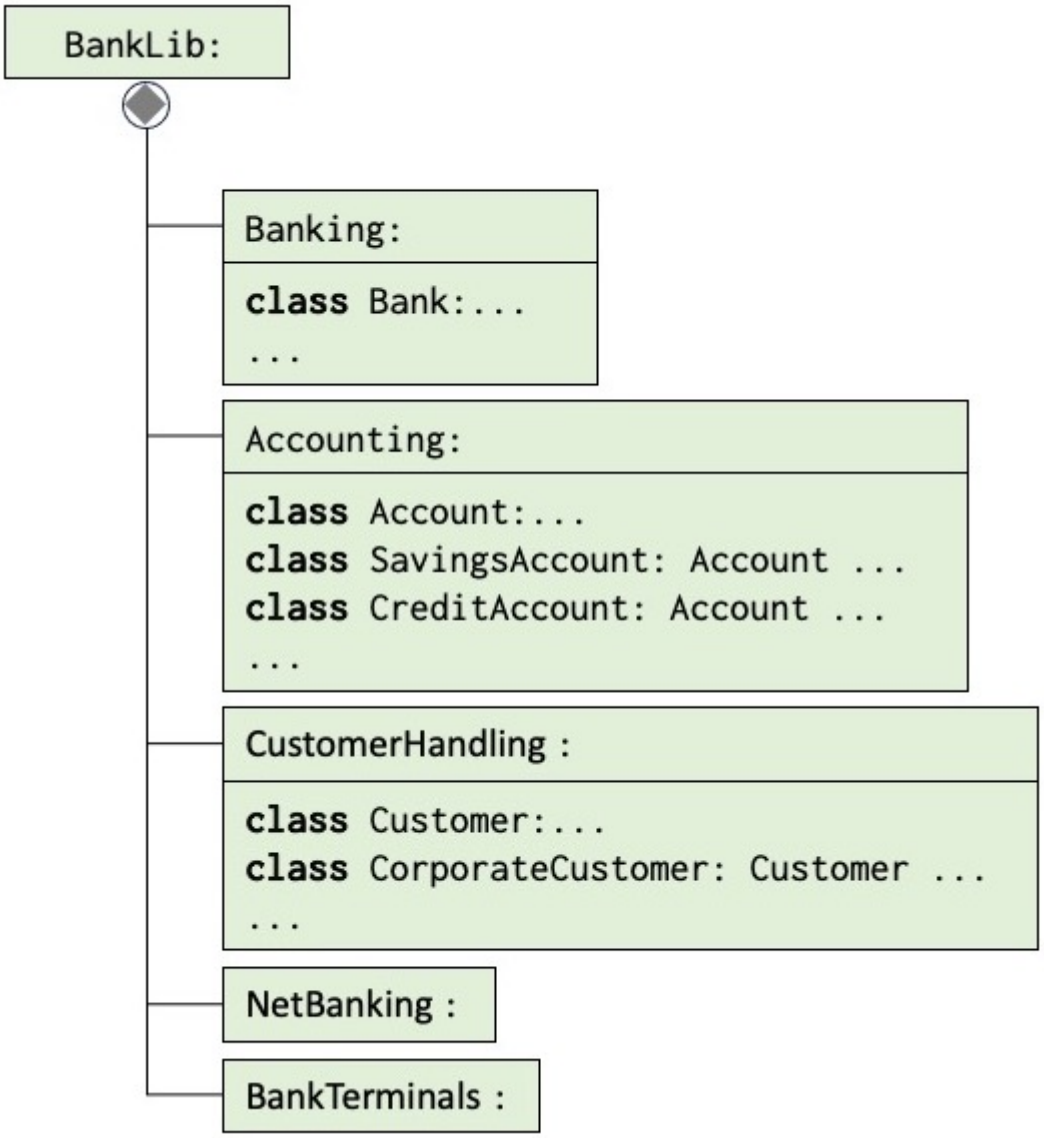
The example makes use of class `Grammar`, which is defined in a module `GrammarLib`. It has a local module `demo`, which has `ExpressionGrammar` as a local module. The expression grammar has local modules `expAst` and `ExpParser` that contain the abstract syntax tree of the grammar and the parser for the grammar. This module hierarchy is shown in the diagram.



Extending BankLib

Next we show an extension of `BankLib` where we add additional modules:

```
BankLIB: obj
   Banking: obj
      class Bank: ...
      ...
   Accounting: obj
      class Account: ...
      class SavingsAccount: Account ...
      class CreditAccount: Account ...
      ...
   CustomerHandling: obj
     class Customer: ...
     class CorporateCustomer: Customer ...
     ...
   NetBanking: obj
     ...
   BankTerminals: obj
     ...
```



Organizing BasicSystem and MonitorSystem

Another example of organizing modules is the organization of `BasicSystem` and `MonitorSytem`. They may both be placed as modules in `LIB`:

```
qBetaWorld: obj
   qBeta: obj
      ...
   LIB: obj
      BasicSystemLib: obj
         -"-
      MonitorSystemLib: obj
         class MonitorProcess:
            -"-
         class Monitor:
           -"-
      ...
   -"-
  Workspace: obj
      aSafeSearcher: obj LIB.MonitorSystemLib.MonitorSystem
```

```
        searcherA: Searcher -"-
        -"-
        collector: obj Monitor -"-
        presenter: obj MonitorProcess -"-
        searcherA.start
        -"-
```

As an example, we have placed `aSafeSearcher` as a module/program in `workspace`.

Example of using the package access modifiers

In the next example we show how to use the package acces modifiers for controlling visibility of attributes within the `BankSys` package.

```
BankLIB: obj
   %package_boundary   banking: obj class Bank:
         -"-
         %package          theAccountsFile: obj Set(Account)   accounting:
obj      class Account:
          -"-
          %package
          balance,interestRate: var float
      class SavingsAccount: Account ...
      class CreditAccount: Account ...      ...
   customers/customerHandling: obj      class Customer: ...
      class CorporateCustomer: Customer ...      ...   NetBanking: obj
      ...
   BankTerminals: obj
      ...
```

- The access modifier `%package_boundary` defines `BankLib` as the top module of the package.
- The `%package` modifier specifies that the data-items `balance` and `interestRate` of `Account` objects are visible in the whole `BankLib` package. Please be aware that thesis just an example – in a real banking system it may not be a good idea to make such attributes visible to the whole package.
- A `%package` modifier is also used to specify that `theAccountsFile` is visible in the whole package.