DRAFT 2.0 -- not ready.

3. Subsystem Development Paradigm.

3.1. General.

While Ada and the Rational Programming Environment can support a wide range of programming methodologies and project management strategies, the language and the environment are particularly suited to those methodologies based upon techniques such as hierarchical decomposition, object-oriented design, levels of abstraction, information hiding, data abstraction, etc. In this section we introduce the Rational Subsystem Paradigm, which is representative of a family of related methodologies that have been developed over the past decade. The methodology described

```
 SSSS  U   U  BBBB    SSSS  Y   Y   SSSS
S      U   U  B   B  S      Y   Y  S
S      U   U  B   B  S       Y Y   S
 SSS   U   U  BBBB    SSS     Y     SSS
    S  U   U  B   B      S    Y        S
    S  U   U  B   B      S    Y        S
SSSS   UUUUU  BBBB   SSSS     Y    SSSS




TTTTT  X   X  TTTTT          333    999    666
  T    X   X    T           3   3  9   9  6
  T     X X     T               3  9   9  6
  T      X      T              3    9999  6666
  T     X X     T               3      9  6   6
  T    X   X    T      ..   3   3      9  6   6
  T    X   X    T      ..    333    999    666
```

```
*START* Job SUBSYS Req #977 for EGB    Date 29-Apr-85   9:41:11 Monitor: //, TOPS
File RM:<MTD.SPEC>SUBSYS.TXT.396, created: 17-Mar-85 19:28:52
        printed: 29-Apr-85  9:41:13
Job parameters: Request created:29-Apr-85  9:38:03   Page limit:54   Forms:NORMAL
File parameters: Copy: 1 of 1   Spacing:SINGLE   File format:ASCII   Print mode:ASC
```

```
 SSSS  U   U  BBBB    SSSS  Y   Y   SSSS
S      U   U  B   B  S      Y   Y  S
S      U   U  B   B  S       Y Y   S
 SSS   U   U  BBBB    SSS     Y     SSS
    S  U   U  B   B      S    Y        S
    S  U   U  B   B      S    Y        S
SSSS   UUUUU  BBBB   SSSS     Y    SSSS




TTTTT  X   X  TTTTT          333    999    666
  T    X   X    T           3   3  9   9  6
  T     X X     T               3  9   9  6
  T      X      T              3    9999  6666
  T     X X     T               3      9  6   6
  T    X   X    T      ..   3   3      9  6   6
  T    X   X    T      ..    333    999    666



*START* Job SUBSYS Req #977 for EGB    Date 29-Apr-85  9:41:11 Monitor: //, TOPS
File RM:<MTD.SPEC>SUBSYS.TXT.396, created: 17-Mar-85 19:28:52
        printed: 29-Apr-85  9:41:13
Job parameters: Request created:29-Apr-85  9:38:03   Page limit:54   Forms:NORMAL
File parameters: Copy: 1 of 1   Spacing:SINGLE   File format:ASCII   Print mode:ASCI
```

DRAFT 2.0 -- not ready.

3. Subsystem Development Paradigm.

3.1. General.

While Ada and the Rational Programming Environment can support a wide range of programming methodologies and project management strategies, the language and the environment are particularly suited to those methodologies based upon techniques such as hierarchical decomposition, object-oriented design, levels of abstraction, information hiding, data abstraction, etc. In this section we introduce the Rational Subsystem Paradigm, which is representative of a family of related methodologies that have been developed over the past decade. The methodology described here is tailored to Ada and the R1000, and provides a framework that can be adapted or extended to address the requirements of a particular project.

We will introduce both the methodology and associated programming environment support by considering key activities in various phases of the development life cycle. For purposes of discussion, we present a very simple view of the development cycle. In practice, development will be very iterative, and at different levels the same software will be in all of the phases described here. Thus, development activities overlap and the facilities discussed find use in every phase of development. We will consider only the following phases:

a. Preliminary design.
b. Detail design and implementation.
c. Test and integration.
d. Maintenance and on-going development.

Note that we do not address requirements analysis, but begin with design tasks. After considering these phases we will briefly consider development in a distributed environment and support for multiple targets, two topics which will be addressed more fully later.

3.2. Preliminary Design.

3.2.1. Decomposing a System into Subsystems.

In the subsystem methodology, a large system is decomposed into a hierarchy of subsystems. For the moment, we will view each subsystem simply as a collection of one or more Ada packages which implement some portion of the system. The system should be decomposed in accordance with good design practices and software engineering principles. For decomposing large Ada systems into subsystems, it is particularly important to recognize two dimensions of decomposition. In the "vertical" dimension, it is important to decompose the system, or any portion of the system, into levels of abstraction, with separate subsystems for major layers. This layering results in a more manageable and maintainable system. In the "horizontal" dimension, the system (at any particular level of abstraction) should be modularized into logical entities, preferrably in an object-oriented manner. Essentially, the decomposition of a large system into subsystems is an extension of the process of decomposing a large Ada program into packages.

In addition to reflecting good design practices, the decompostion of a system into subsystems must reflect organizational and project management considerations. For many projects, a subsystem will correspond to the amount of work that can be reasonably allocated to a single person, or to a small team. Distribution of activities between different development groups with differing expertise (and perhaps different geographic locations) may also influence the decomposition. If the system is to be bundled and unbundled in different product configurations, that separation should be reflected in the subsystem structure. Generally, a subsystem will serve as the field replacable unit for purposes of software repair, release and distribution. Other organizational constraints on system decomposition will vary according the to particular project and development team.

The decomposition into subsystems must identify the subsystems, define (at least at a high level) the contents of each subsystem and specify the interfaces between subsystems. Then the design effort can focus on the individual subsystems, although there will continue to be some evolution of the system structure as the design matures.

3.2.2. Subsystem Interfaces.

A subset of the packages in a subsystem will be exported. The visible parts of all the exported packages form the abstract interface which the subsystem presents to higher-level subsystems. This abstract interface should hide implementation details from higher-level subsystems, while completely capturing the facilities to be provided by the subsystem. Again, good design practices based on information hiding, abstraction, etc., should be applied in designing subsystem interfaces.

A subsystem will import lower-level subsystems to use in its implementation. This "using" relationship, where one subsystem uses another in its implementation or it Ada specs, must form a strict hierarchy (no cycles).

3.2.3. Subsystem Design.

Once the interfaces have defined, it is possible to design the subsystem itself. The design of the individual subsystem should conform to good software engineering practices, but is largely driven by the specific application and the system design goals. Each subsystem should be designed to be indepently tested and maintained to the greatest extent possible.

Precisely specifying the abstract interface for a subsystem and then constructing the subsystem on top of other subsystems, brings us to the next phase of development.

3.3. Detailed Design and Implementation.

3.3.1. Subsystems as R1000 Control Points.

In the environment, each subsystem is represented as an object control point (see 2.3.1). The control point contains the specs which make up the abstract interface for the subsystem, contains the bodies for all those specs, contains other library units required to implement the subsystem, and may contain any managed objects which store information relating to the subsystem. The use of a control point as a subsystem exploits the control point configuration mechanism.

3.3.2. Subsystem Releases.

Each version of the control point configuration (see 2.3.4) represents a consistent view of the subsystem, which we will call a release.

3.3.3. Creating a Subsystem.

Consider the example of creating a new subsystem named Foo. This in turn creates an initial (fairly empty) release. In creating the control point we had the option of specifying the name of the first release. Let us assume we named it F_0_0, representing the first release of the subsystem Foo. We can specify that the F_0_0 release of Foo is part of our session configuraiton. Operations on any control point are with respect to the specified release. For example, the first operation on the newly created library might be to create several new Ada units. This would update the release to reflect the new objects (and the current versions of those objects).

3.3.4. Compilation and Semantic Consistency.

In accordance with section 2.3.4., each release includes compilation switches and a target key. For our example, assume that we have let the target key default to the R1000. The switches control compilation options, and are passed to any compilation that occurs in the control point. Switches can be set on a per release basis. Consistent with section 2.5, we can compile, elaborate and execute units in the subsystem. The configuration mechanism allows the system to view the release as a single set of units, and ignore version issues. The compilation facilities (both interactive and batch) use the session configuration to determine release of the subsystem, and then compile with respect to that release. As discussed in 2.5.3, the system automatically maintains semantic consistency within a release.

Assume that in our example subsystem, Foo, we have constructed three packages, A, B and C. Further assume that the bodies of these packages reference another subsystem Bar. Having created these units and updated the library context clause, our example release will look like the following.

```
    Library Foo is
        package A is separate;
        package B is separate;
        package C is separate;
        package body A is separate;
        package body B is separate;
        package body C is separate;
```

```
    end Foo;
```

The release configuration includes the newly created versions of each of the packages. However, compiling the subsystem at this point would be only partially successful. The visible parts will all compile, but the bodies reference the subsystem Bar, which must first be imported.

3.3.5. Importing Subsystems.

For casual libraries, the library context clause (see 2.4.2) and normal configuration defaulting mechanisms (see 2.3) are adequate. However, when constructing subsystems it is important to bind the subsystem to a particular version of the abstract interface for each lower-level subsystem that is imported. For this purpose, there is a subsystem import operation which updates the library context clause as necessary, and updates the release configuration to reflect the specified release of the imported subsystem.

In the example we have been using we might wish to import the B_0_1_5 release of Bar, which is an export release of Bar that includes all of the specs we will need in implementing Foo. Construction of export releases is discussed later. The import operation would update the library context for Foo to include Bar and would update the release configuration to include all of the specs for B_0_1_5. In general, a release (control point) configuration identifies a version of each unit in the subsystem, and a version of each spec imported into the subsystem. Subsystems never rely upon defaulting mechanisms to access imported units, but rely upon explicit importing.

Having completed the import operation, compilation of Foo will proceed properly.

3.3.9. Local and Global Diana tools.

There are two classes of tools that use Diana on the system, local tools and global tools. The first class relies only upon the release configuration and references only units within the control point and imported specs. The compiler is probably the most important member of this first class. The second class uses a system configuration to look through references to specs to reach the "real" version of the particular unit. The editor and debugger are members of the second class. (Nine out of ten respondents disagree with have the editor automatically look through specs. Still working on that one.)

In this model, compilation is more efficient, but relies only upon local information. Debugging and editing make less frequent use of semantic attributes, but provide a complete and consistent view of a more global universe. For example, using DEFINITION in the editor will take the user to the real spec, so that he may than use OTHER_PART to see the body. The user would only see the (truncated) specs in the spec sublibraries in the case where his session configuration does not include a real version of the referenced subsystem.

One can proceed in this manner, designing and implementing the subsystem as a consistent set of Ada units. As soon as a first (possibly incomplete) release of the subsystem has been compiled, one will want to test and debug that release before proceeding to add functionality or otherwise change the subsystem. The issues of execution, testing and debugging are addressed in the following section.

3.4. Test and Integration.

We will continue the example of the previous section to illustrate the model for test and integration. Let us assume that the Foo subsystem we have constructed is to be a SHARED_ONLY subsystem, and that the Bar subsystem is a SHARED_ONLY subsystem which the owner has already elaborated (i.e., our session configuration references an elaborated release of Bar). Let us further assume that our initial test plan involves the following steps:

Step 1. Construct a Foo Test library which will hold test programs as we write them.

Step 2. Execute several simple commands which exercise facililites from Bar that the Foo needs.

Step 3. In the Foo Test library, construct a more comprehensive test which excercises key Bar facilities in a manner similar to actual foo operation.

Step 4. Elaborate the Foo subsystem, using the debugger as needed to analyze problems.

Step 5. Having successfully elaborated the Foo, execute several simple commands which provide a preliminary indication that the foo has initialized itself properly.

Step 6. In the Foo Test library, construct a more comprehensive test which verifies proper foo initialization checks correct operation of simple facilities.

Step 7. The owner of Bar has produced a new release and would like to test his subsystem using the Foo and Foo Tests procuced earlier.

These steps are only a few of the many required, but they illustrate key characteristics of the subsystem paradigm. Even before discussing the individual steps, it is clear that the R1000 supports an interactive and incremental approach to test and integration.

3.4.1. Constructing a Test Library.

First we construct our test library nested within the foo library. This test library will consists of test programs which execute on top of the exported Foo specs. Thus the Foo Library now looks like the following.

```
    use Bar;
    Library Foo is
        package A is separate;
        package B is separate;
        package body A is separate;
        package body B is separate;
        library Test is separate;
```

```
    end Foo;
```

3.4.2. Establishing the Test Configuration.

Step 2 in our plan involves executing a few simple commands to exercise facilities in Bar that the Foo would rely upon. All execution requires a system configuration, and in this case our session configuration is adequate since it should include an elaborated Bar. The key issue here is that we must have properly established our session configuration so that it includes an elaborated release of Bar which supports the facilities we need for testing the Foo.

This may be an unnecessary step, but prevents wasting time because of improperly established test configurations. While the subsystem paradigm is designed to help eliminate many of these problems, cross subsystem coordination requires manual intervention and is subject to some error. Therefore, the system facilitates quick, interactive verification at key steps, so that errors which do occur can be detected early in the process. Once the user is confident that his session configuration is being established properly, this step may be eliminated.

Recall that for SHARED_ONLY libraries the library is elaborated as a whole. If we visit one of the specs in Bar, it should be in the elaborated state at this point. If it is not, we must (possibly in cooperation with the owner of Bar) either update our configuration to reflect an already elaborated release of Bar, or create an elaborated release for our own use.

We can then write short commands which call specs exported by machine interface. Even if these test commands do not exercise the most interesting facilities, they give us quick feedback that Bar is properly elaborated. If there are new facilities that have just been added for our use, we might try to exercise those to make sure they at least exist. If any errors are uncovered, we can construct minimal test cases which produce the problem, and then work with the owner of Bar to resolve the problems.

3.4.3. Constructing a Test Program.

Step 3 is a continuation of Step 2. While Bar presumably has a set of test programs, we may want a test program that further verifies specific properties that we depend upon. We add this to the Foo Test library and compile against the specs imported from Bar. The test library is not a shared library, and we can call the test program as soon as it has been coded. The library Foo.Test would look like the following at this point.

```
    use Bar;
    Library Test is
        procedure Test_KMI is separate;
    end;
```

From a command window we can execute Test_KMI and review the results. If the test produces a log file we can save a "golden" copy of the file in the test library, and have each execution compare its results to the golden results. We can later add test drivers in the test library which invoke this test along with others and produce a summary of the results.

3.4.4. Elaboration of a new Subsystem.

Step 4 actually involves executing the new code we have written in the Foo library. We can elaborate the current release of the Foo, or we can produce a new release of the Foo which differs from the previous only in that we will promote it to elaborated. Elaboration information is retained as part of the release automatically. When elaborating a SHARED_ONLY subsystem we have the following three options with respect to the persistence of the elaboration.

1. The subsystem remains elaborated from the time it is explicitly elaborated until it is demoted or until the system goes down, whichever comes first.

2. The subsystem remains elaborated until it is demoted, and it is automatically elaborated after a crash at the time the system is brought up.

3. The subsystem remains elaborated until it is demoted, and it is automatically elaborated after a crash at the time of the first reference to the subsystem that requires it to be elaborated.

While we are first debugging the subsystem, option 1 is most appropriate, since we do not expect anyone else to be using the subsystem. Once we have released a version of the Foo for widespread use, we must chose between option 2 and option 3. The system elaborates the new subsystem with respect to a particular system configuration, in this case our session configuration. Since the constructions of any system configuration verifies subsystem compatibility, we are certain that we are elaborating against compatible versions of the lower level subsystems. This consistency checking is addressed further in 3.5., since the main issues deal with upward compatible changes.

If any problems are encountered with the elaboration of the Foo (quite likely if there is much new code), we can use the interactive debugging facilities to investigate the problems. Interactive debugging is further discussed in section xxxx. Once we have successfully elaborated the subsystem, we can move on to the next phase of testing.

4.4.5. Test and Release of a new Subsystem.

Step 5 in our test plan involves executing a few commands to check that the newly elaborated Foo is properly elaborated and that basic facilities work properly. This gives us quick feedback and lets us interact with the subystem directly to determine its health. If the Foo exports operations which perform internal consistency checks, those are probably the first operations we invoke. From a command window, we can invoke any operation exported by any package in the subsystem, including exported packages (Foo.A and Foo.B) and internal packages (Foo.C). Having executed some of the visible operations, perhaps with the debugger, we move on to the next step.

Step 6 involves adding to the Foo Test library a more comprehensive test of the initialization of the Foo. The test library would now look like the following.

```
    use Bar;
    use Foo;
    Library Test is
        procecure Test_KMI is separate;
        procecure Test_Initialization is separate;
    end;
```

Once this test program has been coded, we can execute it and determine the results. Again this test can be structured so that the results are reproducibly verifiable. We can continue in this vein, executing simple commands that excercise the Foo directly, writing more test programs, and building up our test library, and then cycling back to design and implementation of additional Foo facilities.

In preparation for moving on to step 7, let us assume that these first two test programs work correctly abd we and freeze an elaborated release of the Foo and an unelaborated release of Foo.Test. We can establish these releases as the defaults, which other users will see if they do not specify particular releases. In a very minimal sense, the subsystem has been released. A more formal release process can be supported, including more comprehensive test and documentation procedures. For the moment, assume we are constructing an informal, internal release.

3.4.6. Recombinant test and integration.

Now assume that while we were developing the Foo, the owner of Bar has just frozen a new release of his subsystem. In addition to running his own regression tests, he now has a customer (the Foo) who has code which executes on top of Bar. He may wish to run the Foo tests, since they may actually exercise Bar in more or different ways.

The subsystem paradigm allows the combination of different versions of subsystems that have compatible interfaces. Thus the owner of Bar should be able run his new subsystem with the previously released Foo, which is known to execute properly with the previous release of Bar. This property will hold for the R1000 as an execution vehicle in the face of a fairly wide range of upward compatible changes in the different versions of the subsystem specs (including different private parts, adding new functions, etc., see 3.5.). Non-R1000 targets which follow fairly simple conventions for linking and loading may also be able to support this aspect of the subsystem paradigm, although a smaller (possibly empty) set of upward compatible changes will be supported (see 2.5.).

In our example, the session configuration for the owner of Bar currently includes the new release of Bar. We have released Foo and Foo.Test libraries, so he can get those releases automatically. However, in this case he cannot add the elaborated release to his session, because it is elaborated against a different Bar. However, if he designates the Foo subsystem and demotes it to coded, he implicitly spawns a new release (which is added to his session configuration), which can be elborated on top of his new

Bar. Then he can execute the Foo tests and ensure that the new release of Bar supports the current release of the Foo. He may even include this as part of the standard regression testing procedure for new releases of Bar.

Note that relatively little interaction is required between the developers of different subsystems. There may be many releases of Bar or many releases of the Foo, but they need not be coordinated as long as they are spec compatible. In practice, interface issues or subtle bugs may arise, requiring joint debugging and coordinated fixes.

So far we have focused on testing an individual subsystem in the context of other subsystems. System testing can simply be viewed as testing the "top" subsystem in terms of all the lower subsystems. For system testing, and even subsystem testing, one can create configuration objects which capture meaningful configurations of subsystems. The system will enforce consistency and ensure that the systems are configured properly.

More on regression testing, system test, test tools, etc. Someday.

3.5. Maintenance and On-Going Development.

So far we have been considering very simple scenarios involving new subsystems with very few versions. Much more complex issues arise during maintenance and on-going development of a subsystem which has one or more released version which must be supported. In particular, support for incremental and upward-compatible changes becomes essential (since one is "fixing" existing code rather than writing new code), source managment becomes a major issue, and tracking of history and other information becomes more important. Let us continue with the example of previous sections to illustrate these issues.

3.5.1. Incremental and Upward Compatible Changes.

Recall that at the end of step six in the previous section we froze a version of the Foo. Now assume that based on our first round of incremental testing we wish to fix several problems and add several facilities in the package Foo.C. We spawn a new release, KK_0_1. All of the units are initially shared with the previous release (KK_0_0). If we make changes in package C, only that package will have a new version which is in the new Foo release and not in the old release. The Foo subsystem configuration we are using is updated to reflect these new versions of package C. When we have made our changes, probably using the incremental compilation facilities, we can test them immediately by elaborating this new release and repeating a form of the test cycle described in the previous section.

Changing exported specs provides a more interesting example. Assume that several clients are now using the frozen version of Foo released earlier (KK_0_0), and we wish to change exported Foo specs to produce a new release (KK_0_1) that is compatible with the code the clients have produced, but which includes new facilities that support future client development. In particular, we wish to be able to have the clients run against

KK_0_1 (when we release it) without the clients having to recompile any of there subsystems. In fact, old frozen versions of the clients which ran against KK_0_0 should run against KK_0_1, while new versions of client code may be developed using the new facilities of KK_0_1.

The system actually goes further, in that clients can import the new specs into their subsystems without causing any recompilation. The only compilation involved is that associated with changes the client might make to use the new facilities provided by the new foo specs.

We cannot demote the specs to source and make arbitrary changes and have the changes be upward compatible. However, if we make incremental changes, the system will produce a new version (reflected in the KK_0_1 subsystem configuration) and properly maintain the version so that it is upward compatible. In particular, it will not allow incremental deletions without informing us that such a change would not be upward compatible. Incremental additions would be supported, and the system would properly maintain semantic and code generator attributes such that the change is upward compatible.

For the R1000, most additions are upward compatible, and if we have specifiec that the spec has a closed private part, then no client was allowed to rely upon the information in the private part and all changes which affect only the private part are then upward compatible. (The complete set of rules for upward compatibility for each target (including the R1000) will specified separately.)

The system implements upward compatibility by restricting the set of changes allowed in the spec, and managing specific compiler attributes. Proper management of compiler attributes requires that all compatible versions of a spec be maintained (especially modified) on a single machine which maintains the set of related versions of the specs. Each compatible version of a spec is related to the original and shares a compatibility key which is used for cross-subsystem compatibility checking. Anytime a system configuration is constructed, exported specs are checked against imported specs (for all subsystems in the configuration) to ensure that they have the same compatibility key.

In our example, we could add changes to Foo.A and Foo.B (exported packages) using the incremental facilities of the environment, and then test those changes as described above. Once we are satisfied that the changes have been made properly, we can freeze and release KK_0_1. If we make that release the new default, and encourage clients update their system configurations properly, then users will be using KK_0_1, which should support all old facilities, plus the new facilities we have provided.

3.5.2. Source Management.

The facilities discusses so far are adequate for supporting a single development path, where there is a sequence of releases, each release superceding the previous release. Given that the system has been decomposed into small subsystems where a very small team is working on each subsystem, and given that only a single development path need be supported, no additional source management support would be required. However, in the face of maintaining one or more released versions while supporting one or more active new development paths involving more people and possibly multiple target machines, the user will required more substantial source management support.

basic model is to know which devel paths are related, inform user when he is making a change that it will impact other paths, support policies that restrict changes which impact other paths, and support merging and (semi)automatic propogation of changes to other paths.

source mgmt -- serial releases, divergence and parallel devel, multi (lower-level) specs and multi targets.

3.5.3. History.

keep all relevant info at object, release, subsystem, and project level. support construction of tools that operate on this info.