Overview

Introduction

Origins and Purposes of Projects

- Need for flexibility
  - Managing huge applications is a difficult task
  - Build tools are always useful
- GNAT compilation model
  - Compiler needs to know where to find Ada files imported by Ada unit being compiled
- IDEs
  - AdaCore IDEs need to know where to find source and object files
- Tools (metrics, documentation generator, etc)
  - AdaCore Tools benefit from having knowledge of application structure

Subsystems of Subsystems of ...

- Projects support incremental, modular project definition
  - Projects can import other projects containing needed files
  - Child projects can extend parent projects
    - Inheriting all attributes of parent
    - Can optionally override source files and other attributes
- Allows structuring of large development efforts into hierarchical subsystems
  - Build decisions deferred to subsystem level

Project Files

GNAT Project Files

- Text files with Ada-like syntax
- Also known as *GPR files* due to file extension
- Integrated into command-line tools
  - Specified via the `-P project-file-name` switch
- Integrated into IDEs
  - A fundamental part
  - Automatically generated if desired
- Should be under configuration management

Configurable Properties

- Source directories and specific files' names
- Output directory for object modules and .ali files
- Target directory for executable programs
- Switch settings for project-enabled tools
- Source files for main subprogram(s) to be built
- Source programming languages
  - Ada / C / C++ are preconfigured
- Source file naming conventions
- et cetera

The Minimal Project File

```
project My_Project is
end My_Project;
```

Specifying Main Subprogram(s)

- Optional
  - If not specified in file, must be specified on command-line
- Can have more than one file named
- A project-level setting

```
project Foo is
   for Main use ("bar.adb", "baz.adb");
end Foo;
```

About Project Files and Makefiles

- A Makefile performs actions (indirectly)
- A project file describes a project
- Command lines using project files fit naturally in Makefile paradigm

```
gprbuild -P <project-file> ...
```

Building with GPRbuild

Introduction

Generic Build Tool

- Designed for construction of large multi-language systems
  - Allows subsystems and libraries
- Manages three step build process:
  - Compilation phase:
    - Each compilation unit examined in turn, checked for consistency, and, if necessary, compiled (or recompiled) as appropriate
  - Post-compilation phase (binding):
    - Compiled units from a given language are passed to language-specific post-compilation tool (if any)
    - Objects grouped into static or dynamic libraries as specified
  - Linking phase:
    - Units or libraries from all subsystems are passed to appropriate linker tool

Command Line

GPRbuild Command Line

- Made up of three elements
  - Main project file (required)
  - Switches (optional)
    - `gprbuild` switches
    - Options for called tools
  - Main source files (optional)
    - If not specified, executable(s) specified in project file are built
    - If no main files specified, no executable is built

Common Options Passed To Tools

- `-cargs options`
  - Options passed to all compilers
  - Example:

    ```
    -cargs -g
    ```
- `-cargs:<language> options`
  - Options passed to compiler for specific language
  - Examples:

    ```
    -cargs:Ada -gnatf
    -cargs:C -E
    ```
- `-bargs options`
  - Options passed to all binder drivers
- `-bargs:<language> options`
  - Options passed to binder driver for specific language
  - Examples:

    ```
    -bargs:Ada binder_prefix=ppc-elf
    -bargs:C++ c_compiler_name=ccppc
    ```
- `-largs options`
  - Options passed to linker for generating executable

Common Command Line Switches

`-P <project file>`	Name of main project file (space between *P* and *<filename>* is optional)
`-aP <directory>`	Add *<directory>* to list of directories to search for project files
`-u [<source file> [, <source file>...]]`	If sources specified, only compile these sources. Otherwise, compile all sources in main project file
`-U [<source file> [, <source file>...]]`	If sources specified, only compile these sources. Otherwise, compile all sources in project tree
`-Xnm=val`	Specify external reference that may be read via built-in function `external`.
`--version`	Display information about GPRbuild: version, origin and legal status
`--help`	Display GPRbuild usage
`--config=<config project file name>`	Configuration project file name (default `default.cgpr`)

Common Build Switches

Switches to be specified on command line or in `Builder` package of main project file

`--create-map-file[=<map file>]`	When linking, (if supported) by the platform, create a map file `<map file>`. (If not specified, filename is `<executable name>.map`)
`-j<num>`	Use <num> simultaneous compilation jobs
`-k`	Keep going after compilation errors (default is to stop on first error)
`-p (or --create-missing-dirs)`	Creating missing output directory (e.g. object directory)

Lab

Start GPRBuild

- Open a command shell
- Go to `020_building_with_gprbuild` directory (under `source`)
  - Contains a main procedure and a supporting package for the "8 Queens" problem
- Use an editor to create minimum project file
  - Name the project anything you wish
  - Filename and project name should be the same
- Build `Queens` using `gprbuild` and the project file as-is
  - Use `-P` argument on the command line to specify project file
  - Must also specify file name on command line to get executable
    - For example: `gprbuild -P lab.gpr queens`
- Clean the project with `gprclean`
  - Use `-P` argument on the command line to specify project file
  - Note that the `queens.exe` executable remains
    - Plus (possibly) some intermediate files

GPRbuild Lab - Simple GPR File

```
project Lab is
end Lab;
```

`gprbuild -P lab.gpr` Only compiles source files

`gprbuild -P lab.gpr queens` Compiles source and creates `queens` executable

`gprclean -P lab.gpr` Deletes ALI and object files for `Queens` and `Queens_Pkg`

GPRbuild Lab Part 2

- Change project file so that it specifies the main program
- Build again, without specifying the main on the command line
  - Use only `-P` argument on the command line to specify project file
- Clean the project with `gprclean` again
  - Note the `queens` executable is now also deleted (as well as any intermediate files)

GPRbuild Lab - Main Program Specified

```
project Lab is
   for Main use ( "main.adb" );
end Lab;
```

`gprbuild -P lab.gpr` Compiles source and creates `queens` executable

`gprclean -P lab.gpr` Deletes all generated files

Project Properties

Introduction

Specifying Directories

- Any number of Source Directories
  - Source Directories contain source files
  - If not specified, defaults to directory containing project file
  - Possible to create a project with no Source Directory
    - Not the same as not specifying the Source Directory!
- One Object Directory
  - Contains object files and other tool-generated files
  - If not specified, defaults to directory containing project file
- One Executables Directory
  - Contains executable(s)
  - If not specified, defaults to same location as Object Directory
- *Tip: use forward slashes rather than backslashes for the most portability*
  - Backslash will only work on Windows
  - Forward slash will work on all supported systems (including Windows)

Variables

Typed Set of possible string values

Untyped Unspecified set of values (strings and lists)

```
project Build is
   type Targets is ("release", "test");
   -- Typed variable
   Target : Targets := external("target", "test");
   -- Untyped string variable
   Var := "foo";
   -- Untyped string list variable
   Var2 := ("-gnato", "-gnata");
   ...
end Build;
```

Typed Versus Untyped Variables

- Typed variables have only listed values possible
  - Case sensitive, unlike Ada
- Typed variables are declared once per scope
  - Once at project or package level
  - Essentially read-only constants
    - Useful for external inputs
- Untyped variables may be "declared" many times
  - No previous declaration required

Property Values

- Strings
- Lists of strings

  `("-v", "-gnatv")`
- Associative arrays
  - Map input string to either single string or list of strings

    for <name> (<string-index>) use <list-of_strings>;

    ```
    for Switches ("Ada") use ("-gnaty", "-gnatwa");
    ```

Directories

Source Directories

- One or more in any project file
- Default is same directory as project file
- Can specify additional / other directories

```
for Source_Dirs use ("src/mains", "src/drivers", "foo");
```

- Can specify an entire tree of directories

```
for Source_Dirs use ("src/**");
```

  - `src` directory and every subdirectory underneath

Source Files

- Must be at least one **immediate** source file
  - *Immediate*
    - Resides in project source directories OR
    - Specified through source-related attribute
  - Unless explicitly specified none present

```
for Source_Files use ();
```

- Can specify source files by name

```
for Source_Files use ("pack1.ads","pack2.adb");
```

- Can specify an external file containing source names

```
for Source_List_File use "source_list.txt";
```

Object Directory

- Specifies location for compiler-generated files
  - Such as `.ali` files and object files
  - For the project's immediate sources

```
project Release is
   for Object_Dir use "release";
   ...
end Release;
```

- Only one object directory per project
- If `Child` extends project `Parent` and then building `Child`
  - For any source that exists only in `Parent` but has not been compiled, it's object files will appear in the `Child` object directory

Executable Directory

- Specifies the location for executable image

```
project Release is
   for Exec _Dir use "executables";
   ...
end Release;
```

- Default is same directory as object files
- Only one per project

Project Packages

Packages Correspond to Tools

- Packages within project file contain switches (generally) for specific tools
- Allowable names and content defined by vendor
  - Not by users

- Binder
- Builder
- Check
- Clean
- Compiler
- Cross_Reference
- Documentation
- Eliminate
- Finder
- Gnatls
- Gnatstub
- IDE
- Install
- Linker
- Metrics
- Naming
- Pretty_Printer
- Remote
- Stack
- Synchronize

Setting Tool Switches

- May be specified to apply by default

```
package Compiler is
   for Default_Switches ("Ada") use ("-gnaty", "-v");
end Compiler;
```

- May be specified on per-unit basis
  - Associative array "Switches" indexed by unit name

```
package Builder is
   for Switches ("main1.adb") use ("-O2");
   for Switches ("main2.adb") use ("-g");
end Builder;
```

Naming Considerations

Rationale

- Project files assume source files have GNAT naming conventions
  Specification `<unitname>[-<childunit>].ads`
  Body `<unitname>[-<childunit>].adb`
- Sometimes you want different conventions
  - Third-party libraries
  - Legacy code used different compiler
    - Changing filenames would make tracking changes harder

Source File Naming Schemes

- Allow arbitrary naming conventions
  - Other than GNAT default convention
- May be applied to all source files in a project
  - Specified in a package named `Naming`
- May be applied to specific files in a project
  - Individual attribute specifications

Foreign Default File Naming Example

- Sample source file names
  - Package spec for `Utilities` in **`utilities.spec`**
  - Package body for `Utilities` in **`utilities.body`**
  - Package spec for `Utilities.Child` in **`utilities.child.spec`**
  - Package body for `Utilities.Child` in **`utilities.child.body`**

```
project Legacy_Code is
   ...
   package Naming is
      for Casing use "lowercase";
      for Dot_Replacement use ".";
      for Spec_Suffix ("Ada") use ".spec";
      for Body_Suffix ("Ada") use ".body";
   end Naming;
   ...
end Legacy_Code;
```

GNAT Default File Naming Example

- Sample source file names
  - Package spec for `Utilities` in **`utilities.ads`**
  - Package body for `Utilities` in **`utilities.adb`**
  - Package spec for `Utilities.Child` in **`utilities-child.ads`**
  - Package body for `Utilities.Child` in **`utilities-child.adb`**

```
project GNAT is
   ...
   package Naming is
      for Casing use "lowercase";
      for Dot_Replacement use "-";
      for Spec_Suffix ("Ada") use ".ads";
      for Body_Suffix ("Ada") use ".adb";
   end Naming;
   ...
end GNAT;
```

Individual (Arbitrary) File Naming

- Uses associative arrays to specify file names
  - Index is a string containing the unit name
    - Case insensitive
  - Value is a string containing the file name
    - Case sensitivity depends on host file system
- Has distinct attributes for specs and bodies

for Spec ("<unit name>") use "<filename>";

```
for Spec ("MyPack.MyChild") use "MMS1AF32.A";

for Body ("MyPack.MyChild") use "MMS1AF32.B";
```

Variables for Conditional Processing

Two Sample Projects for Different Switch Settings

```
project Debug is
  for Object_Dir use "debug";
  package Builder is
    for Default_Switches ("Ada")
       use ("-g");
  end Builder;
  package Compiler is
    for Default_Switches ("Ada")
       use ("-fstack-check",
            "-gnata",
            "-gnato");
  end Compiler;
end Debug;
```

```
project Release is
  for Object_Dir use "release";
  package Compiler is
    for Default_Switches ("Ada")
       use ("-O2");
  end Compiler;
end Release;
```

External and Conditional References

- Allow project file content to depend on value of environment variables and command-line arguments
- Reference to external values is by function

  `external (<name> [, default])`

  - Returns value of **name** as supplied via
    - Command line
    - Environment variable
    - If not specified, uses **default** or else ""
- Command line switch

  gprbuild -P... -Xname=value ...

```
gprbuild -P common/build.gpr -Xtarget=test common/main.adb
```

- **Note:** Command line values override environment variables

External/Conditional Reference Example

```
project Build is
   type Targets is ("release", "test");
   Target : Targets := external("target", "test");
   case Target is -- project attributes
      when "release" =>
         for Object_Dir use "release";
         for Exec_Dir use ".";
      when "test" =>
         for Object_Dir use "debug";
   end case;
   package Compiler is
      case Target is
         when "release" =>
            for Default_Switches ("Ada") use ("-O2");
         when "test" =>
            for Default_Switches ("Ada") use
                  ("-g", "-fstack-check", "-gnata", "-gnato");
      end case;
   end Compiler;
   ...
end Build;
```

Scenario Controlling Source File Selection

```
project Demo is
   ...
   type Displays is ("Win32", "ANSI");
   Output : Displays := external ("OUTPUT", "Win32");
   ...
   package Naming is
      case Output is
         when "Win32" =>
            for Body ("Console") use "console_win32.adb";
         when "ANSI" =>
            for Body ("Console") use "console_ansi.adb";
        end case;
   end Naming;
end Demo;
```

- Source Files

`console.ads`	`console_win32.adb`	`console_ansi.adb`
`package Console is`	`package body Console is`	`package body Console is`
`...`	`...`	`...`
`end Console;`	`end Console;`	`end Console;`

Lab

Project Properties Lab

- Create new project file in an empty directory
- Specify source and output directories
  - Use source files from the `030_project_properties` directory (under `source`)
  - Specify where object files and executable should be located
- Build and run executable (pass command line argument of 200)
  - Note location of object files and executable
  - Execution should get `Constraint_Error`

Directories Solution

- Project File

```
project Lab is
   for Source_Dirs use ("source/030_project_properties");
   for Main use ( "main.adb" );
   for Object_Dir use "obj";
   for Exec_Dir use "exec";
end Lab;
```

- Executable Output

```
...
41     267914296
42     433494437
43     701408733
44    1134903170
45    1836311903

raised CONSTRAINT_ERROR : fibonacci.adb:16 overflow check failed
```

Project Properties Lab - Switches

- Modify project file to disable overflow checking
  - Add the `Compiler` package
  - Insert `Default_Switches` attribute for Ada in `Compiler` package
  - Set switch `-gnato0` in the attribute
    - Disable overflow checking
- Build and run again
  - Need to use switch `-f` on command line to force rebuild
    - (Changes to GPR file do not automatically force recompile)
  - No `Constraint_Error`
    - But data doesn't look right due to overflow issues

Switches Solution

- Project File

```
project Lab is
   for Source_Dirs use ("source/030_project_properties");
   for Main use ( "main.adb" );

   package Compiler is
      for Default_Switches ("Ada") use ("-gnato0");
   end Compiler;
   ...
end Lab;
```

- Executable Output

```
...
43    701408733
44   1134903170
45   1836311903
46  -1323752223
47    512559680
48   -811192543
49   -298632863
50  -1109825406
...
```

Project Properties Lab - Naming

- Modify project file to use naming conventions from a different compiler
  - Change source directories to point to `naming` folder
  - File naming conventions:
    - Spec: `<unitname>[.child].1.ada`
    - Body: `<unitname>[.child].2.ada`
  - Remember to fix executable name
- Build and run again
  - *Note: Accumulator uses more bits, so failure condition happens later*

Naming Solution

- Project File

```
project Lab is
   for Source_Dirs use ("source/030_project_properties/naming");

   package Naming is
      for Casing use "lowercase";
      for Dot_Replacement use ".";
      for Spec_Suffix ("Ada") use ".1.ada";
      for Body_Suffix ("Ada") use ".2.ada";
   end Naming;

   for Main use ( "main.2.ada" );
   ...
end Lab;
```

- Executable Output

```
...
88   1779979416004714189
89   2880067194370816120
90   4660046610375530309
91   7540113804746346429
92  -6246583658587674878
93   1293530146158671551
94  -4953053512429003327
95  -3659523366270331776
96  -8612576878699335103
...
```

Project Properties Lab - Conditional

- Modify project file to select precision via compiler switch
  - `conditional` folder has two more package bodies using different accumulators
  - Read a variable from the command line to determine which body to use
    - Hint: `Naming` will need to use a `case` statement to select appropriate body
- Build and run again
  - Hint: Name used in **external** call must be same casing as in GPRBUILD command, i.e
    - `external ("FooBar");` means `gprbuild -XFooBar...`

Conditional Solution

- Project File

```
project Lab is

   for Source_Dirs use ("source/030_project_properties/naming",
                        "source/030_project_properties/conditional");

   type Precision_T is ( "unsigned", "float", "default" );
   Precision : Precision_T := external ( "PRECISION", "default");

   package Naming is
   ...
      case Precision is
      when "unsigned" =>
         for Body ("Fibonacci") use "fibonacci.unsigned";
      when "float" =>
         for Body ("Fibonacci") use "fibonacci.float";
      when "default" =>
         for Body ("Fibonacci") use "fibonacci.2.ada";
      end case;
   end Naming;

   ...
end Lab;
```

- Executable Output

```
1   1.00000000000000E+00
2   2.00000000000000E+00
3   3.00000000000000E+00
4   5.00000000000000E+00
5   8.00000000000000E+00
6   1.30000000000000E+01
7   2.10000000000000E+01
8   3.40000000000000E+01
9   5.50000000000000E+01
10   8.90000000000000E+01
...
```

Structuring Your Application

Introduction

Introduction

- Most applications can be broken into pieces
  - Modules, components, etc - whatever you want to call them
- Helpful to have a project file for each component
  - Or even multiple project files for better organization

Dependency

- Units of one component typically depend units in other components
  - Types packages, utilities, external interfaces, etc
- A project can `with` another project to allow visibility
  - Ambiguity issues can occur if the same unit appears in multiple projects

Extension

- Sometimes we want to replace units for certain builds
  - Testing might require different package bodies
  - Different targets might require different values for constants
- A project can *extend* another project
  - Project inherits properties and units from its parent
  - Project can create new properties and units to override parent

Building an Application

Importing Projects

- Source files of one project may depend on source files of other projects
  - *Depend* in Ada sense (contains `with` clauses)
- Want to localize properties of other projects
  - Switches etc.
  - Defined in one place and not repeated elsewhere
- Thus dependent projects *import* other projects to add source files to search path

Project Import Notation

- Similar to Ada's `with` clauses
  - But uses strings

with <literal string> {, <literal string>};

- String literals are path names of project files
  - Relative
  - Absolute

```
with "/gui/gui.gpr", "../math.gpr";
project MyApp is
   ...
end MyApp;
```

Importing Projects Example

```
with GUI, Math;
package body Pack is
   ...
```

- Source Architecture

/gui		/myapp		/math
gui.gpr	→	myapp.gpr	←	math.gpr
gui.ads		pack.ads		math.ads
gui.adb		pack.adb		math.adb
		main.adb		

- Project File

```
with "/gui/gui.gpr", "/math/math.gpr";
project MyApp is
   ...
end MyApp;
```

Referencing Imported Content

- When referencing imported projects, use the Ada *dot notation* concept for declarations
  - Start with the project name
  - Use the tick (') for attributes

```
with "foo.gpr";
project P is
   package Compiler is
      for Default_Switches ("Ada") use
         Foo.Compiler'Default_Switches("Ada") & "-gnatwa";
   end Compiler;
end P;
```

  - Project P uses all the compiler switches in project Foo and adds `-gnatwa`
  - *Note: in GPR files, "&" can be used to concatenate string lists and string*

Renaming

- Packages can rename (imported) packages
- Effect is as if the package is declared locally
  - Much like the Ada language

```
with "../naming_schemes/rational.gpr";
project Clients is
   package Naming renames Rational.Naming;
   for Languages use ("Ada");
   for Object_Dir use ".";
   ...
end Clients;
```

Project Source Code Dependencies

- Not unusual for projects to be interdependent
  - In the Nav project

```
with Hmi.Controls;
package body Nav.Engine is
   Global_Speed : Speed_T := 0.0;
   procedure Increase_Speed (Change : Speed_Delta_T) is
      Max_Change : Speed_T := Global_Speed * 0.10;
   begin
      Global_Speed :=
        Global_Speed + Speed_T'max (Speed_T (Change),
                                    Max_Change);
      Hmi.Controls.Display_Speed (Global_Speed);
   end Increase_Speed;
end Nav.Engine;
```

  - In the HMI project

```
package body Hmi.Controls is
   procedure Display_Speed (Speed : Nav.Engine.Speed_T) is
   begin
      Display_Speed_On_Console (Speed);
   end Display_Speed;
   procedure Change_Speed (Speed_Change : Nav.Engine.Speed_Delta_T) is
   begin
      Nav.Engine.Increase_Speed (Speed_Change);
   end Change_Speed;
end Hmi.Controls;
```

Project Dependencies

- Project files cannot create a cycle using **with**
  - Neither direct (`Hmi` → `Nav` → `Hmi`)
  - Nor indirect (`Hmi` → `Nav` → `Monitor` → `Hmi`)
- So how do we allow the sources in each project to interact?
  - **limited with**
  - Allows sources to be interdependent, but not the projects

```
limited with "Hmi.gpr";
project Nav is
  package Compiler is
    for Switches ("Ada") use
        Hmi.Compiler'Switches & "-gnatwa"; -- illegal
  end Compiler;
end Nav;
```

Subsystems

- Sets of sources and folders managed together
- Represented by project files
  - Connected by project *with clauses* or project extensions
  - Generally one **primary** project file
  - Potentially many project files, assuming subsystems composed of other subsystems
- Have at most one *objects* folder per subsystem
  - A defining characteristic
  - Typical, not required

Subsystems Example

```
with "gui.gpr";
with "utilities.gpr";
with "hardware.gpr";
project Application is
   for Main use ("demo");
   for Object_Dir use ("objs");
   ...
end Application;

with "utilities.gpr";
project Gui is
   for Object_Dir use ("objs");
   ...
end Gui;

with "utilities.gpr";
project Hardware is
   for Object_Dir use ("objs");
   ...
end Hardware;

project Utilities is
   for Object_Dir use ("objs");
   ...
end Utilities;
```

Building Subsystems

- One project file given to the builder
- Everything necessary will be built, transitively
  - Build `Utilities`
    - Only source specified in `utilities.gpr` will be built
  - Build `Hardware` (or `Gui`)
    - Source specified in `hardware.gpr` (or `gui.gpr`) will be built
    - Source specified in `utilities.gpr` will be built if needed
  - Build `Application`
    - Any source specified in any of the project files will be built as needed

Extending Projects

Extending Projects

- Allows using modified versions of source files without changing the original sources
- Based on *inheritance* of parent project's properties
  - Source files
  - Switch settings
- Supports localized build decisions and properties
  - Inherited properties may be overridden with new versions
- Hierarchies permitted

Limits on Extending Projects

- A project that extends/modifies a project can also import other projects.
- Can't import both a parent and a modified project.
  - If you import the extension, you get the parent
- Can extend only one other project at a time.

Multiple Versions of Unit Bodies Example

- Assume *Baseline* directory structure:
  - `baseline.gpr` contains
    - `filename.ads`
    - `filename.adb`
    - `application.adb`
- For testing, you want to
  - Replace `filename.adb` with a dummy version
  - Use `test_driver.adb` as the main program

Multiple Versions of Unit Bodies Files

- *Baseline* GPR file might look like:

```
project Baseline is
   for Source_Dirs use ("src");
   for Main use ("application");
end Baseline;
```

- Test GPR file might look like:

```
project Test_Baseline extends "Baseline" is
   for Source_Dirs use ( "test_code" );
   for Main use ( "test_driver" );
end Test_Baseline;
```

Lab

Structuring Your Application Lab

- Source is included in folder
  `040_structuring_your_application`
- **Very** simplistic speed monitor
  - Reads current distance
  - Determines amount of time since last read
  - Calculates speed
  - Sends message
- Four subsystems
  - `Base` - types and speed calculator
  - `Sensors` - reads distance from some register
  - `Messages` - sends message to some memory location
  - `Application` - main program
- We could build one GPR file and point to all source directories
  - But as our application grew, this would become harder to maintain

Assignment Part One

1. Build GPR files for each subsystem
   - Hint: These subsystems *depend* on each other, they do not *override* source files
   - As you build each GPR file, run `gprbuild -P <gprfile>` to make sure everything works
   - Main program is in `main.adb`
2. Run `main`
   - This will fail (leading up to Part Two of the assignment)
3. Modify `base_types.ads`
   - Just so source code needs to be compiled
4. Rebuild your main program
   - Even though the modified source file is not directly referenced in the main GPR file, GPRBUILD should compile everything it needs

Assignment Part One - Solution

```
with "../base/base.gpr";
with "../messages/messages.gpr";
with "../sensors/sensors.gpr";
project Application is
   for Source_Dirs use ("src");
   for Object_Dir use "obj";
   for Main use ("main.adb") & project'Main;
end Application;

with "../base/base.gpr";
project Messages is
    for Source_Dirs use ("src");
    for Object_Dir use "obj";
end Messages;

with "../base/base.gpr";
project Sensors is
    for Source_Dirs use ("src");
    for Object_Dir use "obj";
end Sensors;

project Base is
    for Source_Dirs use ("src");
    for Object_Dir use "obj";
end Base;
```

Assignment Part Two

1. Build GPR files to create test stubs for `Odometer` and `Sender`
   - Test bodies exist in the appropriate `test` subfolders
   - Create extensions for `messages.gpr` and `sensors.gpr`
     - We want to inherit the package spec, but use the "test" package bodies
2. Build a GPR file for the main application
   - `Main` still works, we just need the GPR file to access our stubs
   - We could create a new GPR file, or extend the original. Which is easier?
3. Build and run your main program

Assignment Part Two - Solution

- `messages/test` directory

```
project Messages_Test extends "../Messages.gpr" is
   for Source_Dirs use (".");
end Messages_Test;
```

- `sensors/test` directory

```
project Sensors_Test extends "../sensors.gpr" is
   for Source_Dirs use (".");
end Sensors_Test;
```

- `test` directory

```
with "../messages/test/messages_test.gpr";
with "../sensors/test/sensors_test.gpr";
project Test extends "../application/application.gpr" is
   for Main use ("main.adb") & project'Main;
end Test;
```

Advanced Capabilities

Introduction

Other Types of GPR Files

- Project files can also be used for
  - Building libraries
  - Building systems
- Project files can also have children
  - Similar to Ada packages

Library Projects

Libraries

- Subsystems packaged in specific way
- Represented by project files with specific attributes
- Referenced by other project files, as usual
  - Contents become available automatically, etc.
- Library Project

```
library project Static_Lib is
   -- keyword "library" is optional
   ...
end Static_Lib;
```

- Standard Project referencing library

```
with "static_lib.gpr";
project Main is
   ...
end Main;
```

Creating Library Projects

- Several global attributes are involved/possible
- Required attributes

  Library_Name Name of library

  Library_Dir Where library is installed
- Important optional attributes

  Library_Kind *static*, *static-pic*, *dynamic*, *relocatable* (same as *dynamic*)

  Library_Interface Restrict interface to subset of units

  Library_Auto_Init Should autoinit at load (if supported)

  Library_Options Extra arguments to pass to linker

  Library_GCC Use custom linker

Supported Library Types

- Static Libraries
  - Code statically linked into client applications
  - Becomes permanent part of client during build
  - Each client gets separate, independent copy
- Dynamic Libraries
  - Code dynamically linked at run-time
  - Not permanent part of application
  - Code shared among all clients
- Stand-Alone Libraries (SAL)
  - Minimize client recompilations when library internals change
  - Contain all necessary elaboration code for Ada units within
  - Can be static or shared
- See the *GNAT Pro Users Guide* for details

Static Library Project Example

```
library project Name is
   for Source_Dirs use ("src1", "src2");
   for Library_Dir use "lib";
   for Library_Name use "name";
   for Library_Kind use "static";
end Name;
```

- Creates library `libname.a` on Windows

Standalone Library Project Example

```
library project Name is
   Version := "1";
   for Library_Interface use ("int1", "int1.child");
   for Library_Dir use "lib";
   for Library_Name use "name";
   for Library_Kind use "relocatable";
   for Library_Version use "libdummy.so." & Version;
end Name;
```

- Creates library `libname.so.1` with a symlink `libname.so` that points to it

Aggregate Projects

Complex Applications

- Many applications have multiple exectuables and/or libraries
  - Shared source code
  - Multiple "top-level" project files
- Assume project A `withs` project B and project C
  - Build of project A will only compile/link whatever is necessary for project A's executable(s)
  - Executables in project B and C will need to be generated separately
  - Running `gprbuild` on all three projects causes redundant processing
    - Determination of files that need to be compiled
    - Libraries are always built when `gprbuild` is called

Aggregate Projects

- Represent multiple, related projects
  - Related especially by common source code
- Allow managing options in a centralized way
- Compilation optimized for sources common to multiple projects
  - Doesn't compile more than necessary

Aggregate Project Example

```
aggregate project Agg is
   -- Projects to be built
   for Project_Files use ("A.gpr", "B.gpr", "C.gpr");
   -- Directories to search for project files
   for Project_Path use ("../dir1", "../dir1/dir2");
   -- Scenario variable
   for external ("BUILD") use "PRODUCTION";

   -- Common build switches
   package Builder is
      for Global_Compilation_Switches ("Ada")
            use ("-O1", "-g");
   end Builder;
end Agg;
```

Child Projects

Grouping Projects

- Sometimes we want to emphasize project relationships
  - Similar to parent/child relationship in Ada packages
- Child project
  - Declare child of project same as in Ada:
    `project Parent.Child ...`
  - **No inheritance assumed** (unlike Ada)
  - Behavior of child follows normal project definition rules

Child Projects

- Original project

```
-- math_proj.gpr
project Math_Proj is
   ...
end Math_Proj;
```

- Child *depends* on parent

```
with "math_proj.gpr";
project Math_Proj.Tests is
   ...
end Math_Proj.Tests;
```

- Child *extends* parent

```
project Math_Proj.High_Performance extends "math_proj.gpr" is
   ...
end Math_Proj.High_Performance;
```

- Illegal project

```
project Math_Proj.Test is
   ...
end Math_Proj.Test;
```

Summary

Conclusion

GNAT Project Manager Summary

- Supports hierarchical, localized build decisions
- IDEs provide direct support
- GPRbuild allows broad or narrow control over build process
- See the *GPRbuild and GPR Companion Tools User's Guide* for further functionality and capabilities
  - Target build processing
  - Distributed builds
  - Etc