**Incremental Build Process**

The incremental build or partial build is a faster build process where we build only the projects that were modified and its dependent projects as opposed to building all the projects.

***The origins of Incremental build***

To understand the need for incremental build we need to know the drawbacks of full build process:

* Each build takes a long time to complete. (Approximately 1:20 hours)
* Build errors are identified very late in the build process.
* Build logs are more than 100 MB in size and hard to navigate to identify any build issues.

The full build uses full\_build.proj file which contains a list of all the projects in our solution, which are built on each build. We have approximately 270-280 projects in our solution and everything is built for every check-in made by the user, even if the user has made a small change in just one project.

So in incremental build project we try to overcome the drawbacks of full build project by building only the projects that were modified and its dependent project. As a result:

* The average build time of incremental build is about 30 mins.
* As we always build the projects that were modified first, build errors are identified early in the build.

***The Challenge***

So the challenge here is to first identify the projects that were modified. The next task is to identify those projects that are directly dependent on the modified projects.

This task is accomplished in two steps:

1. Identify the dependency between each of our projects.
2. Identify the build order and figure out all the projects that references each of the modified project.

The architecture section below will give more details about the above two steps.

***Architecture***

**Dependency**

**Identifier**

**Dependency**

**Builder**

XML file which lists of all our projects and its references

<Output>

<Input>

full\_build.proj

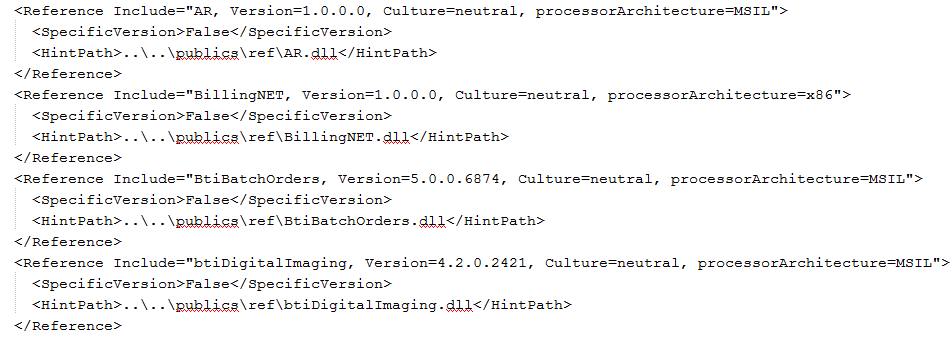
full\_build\_min.proj

<Output>

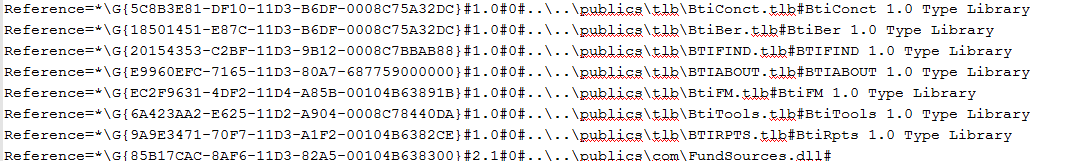
***Dependency Identifier***

Here we parse all the projects in our solution that are listed in the full\_build.proj file. If it’s a .net solution, then we parse the ‘.sln’ file and check for all the projects (.csproj) listed. Then parse each .csproj file to identify all the project reference.

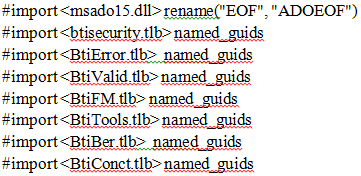
Below is a snapshot of BtiCore.proj file. Here we can clearly see that BtiCore references AR, BillingNet, BtiBatchOrders and other projects.



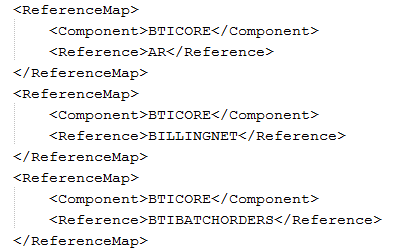
For a visual basic project we parse the ‘.vbp’ file and look for ‘Reference’ to identify other projects that it references. Below is a snapshot of ClientActivity.vbp file, where it’s clearly evident that the lines starting with ‘Reference=’ points to the other projects the Client Activity references.



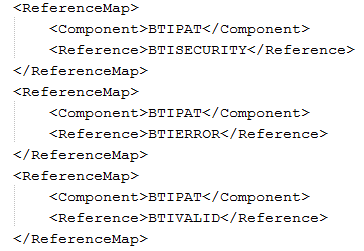
For a vc++ project we parse the ‘StdAfx.h’ file and look for lines beginning with ‘#import’ and ‘#include’ to identify other project references. In the below snapshot of StdAfx.h file in BtiPat, we can clearly identify other projects it imports.



Finally as we identify all these references to other projects it is documented by writing to an xml file (projectReferences.xml). It is saved as a component-reference mapping. Below snapshot shows the component-reference mapping for BtiCore.



Sample BtiPat component-reference mapping:

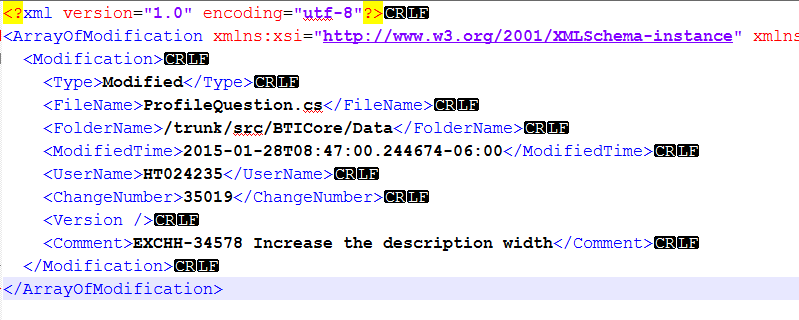


***Dependency Builder***

The Dependency builder project does two things:

1. Generate a sequential order of all our projects including identifying all the circular dependencies.
2. Generate the partial build list.

So the first step in identifying the partial build is to identify the modified projects. On the build machine before every build the cruise control generates the modifications.xml file

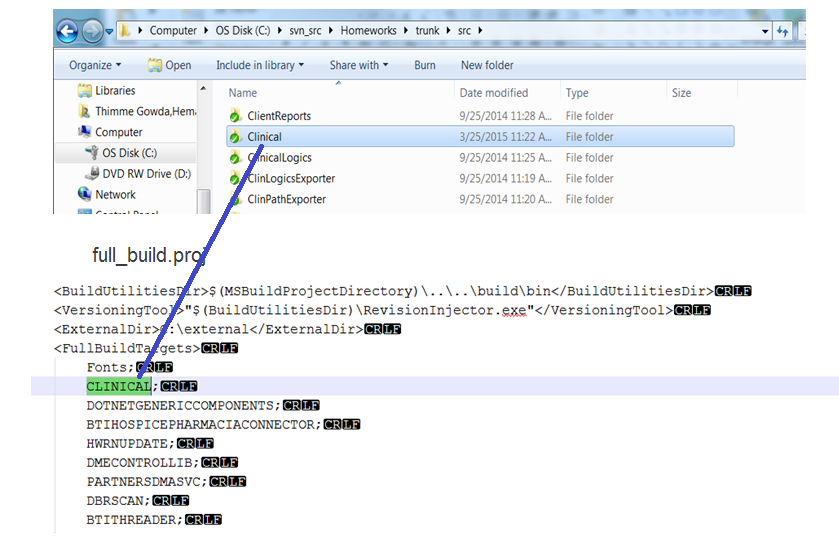


Modifications.xml file holds the information related to the project that is modified or added, and the user who made the change and other details. By parsing this file we can identify what project was modified.

Once we have a list of projects that are modified, we use the component-reference mapping file to identify all those other projects that each of the modified project references. Now the project list in full\_build.proj file should be replaced with the new list of modified and referenced projects. Instead of replacing the full\_build.proj file we create a new file full\_build\_min.proj [essentially a copy of full\_build.proj file but with the project list replaced by the new list of modified and referenced projects]. The full\_build\_min.proj file is used by ‘TrunkPartialBuild’ to do an incremental build.

***Best Practices***

1. Always check-in to public’s folder whenever there is an API change. This was always part of the best practice to have a good build, but with partial build the chances of having a build failure is more if public’s folder is not up to date.
2. Always make sure the project folder name in trunk/src folder match with the project name in full\_build.proj file.



Reason to match the project names: As explained earlier we use modifications.xml file to determine the modified projects. The ‘<FolderName>’ attribute points to the project folder path that was modified.



By parsing the above path and comparing with the list of projects in full\_build.proj file we should be able to identify the modified project. So keeping the project folder name identical for both existing and new projects helps us identify the modified projects.

1. Both full build and partial build are hosted on bti-kruegie. The configurations for both projects are defined in bti\_kruegie\_ccnet.config file.



So whenever we are making a change, please remember the change may be required to both full and partial build configurations.

***Benefits***

1. The average build time of incremental build is much faster than full build. It is atleast 50% faster.
2. Build failures are identified early in the build process. In partial build we first build those projects that were modified, so as a result any code changes that could cause the build to fail are determined early.
3. Incremental build log file is much smaller than full build. It takes less space and faster to navigate.

***Historical Note***

There was no real reason why it was designed as two projects. It would have been better if it was a single project. But the initial reason why we started working on this project was to better understand all the inter-dependencies between all our projects. On that front Matt from TOC has done some work to identify the project dependencies between some of the TOC related projects to get some understanding. Taking that as the base we started working on to extended this to cover all the projects in extended care solution.

Then we created a new project called Dependency Builder which parses the output of Dependency Identifier and generates a sequential list of all the projects such that the project at the beginning of the list has least dependencies and the one after it may be dependent on the project that is earlier in the list. This will help us arrange the projects in full\_build.proj file in proper order and eliminate the dependency on public’s folder. We identified some circular dependencies which made it necessary to have public’s folder and cannot completely eliminate it. However, this did not go completely useless. The work we did here is used to implement incremental build with minor changes.