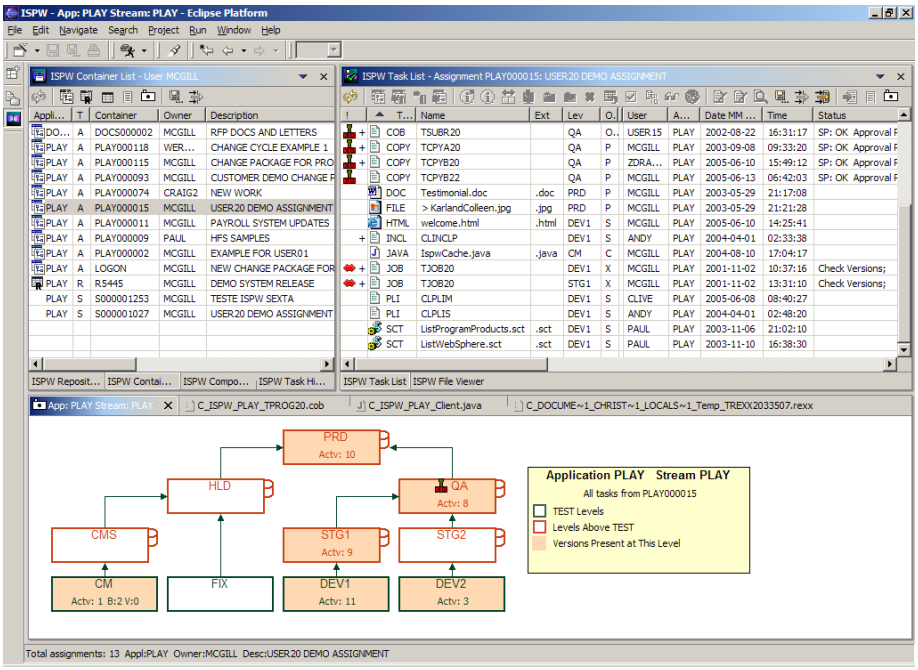


Integrating Software Parts Wherever

Enterprise Software Change Management

ISPW Version 4.2

ISPW History and Architecture - Crash Course

May 2010

ISPW BenchMark Technologies Ltd.

Table of Contents

ISPW Crash Course 1
How ISPW works: *2*

ISPW Crash Course

This is a quick overview of the ISPW product history with a description of how it works.

The ISPW product was originally a large scale application development project at Shell Oil Canada in the mid 1980's. When other companies heard about ISPW at a conference, they asked about licensing it, so ISPW BenchMark was founded in 1986.

Technically, ISPW has been through three complete redesigns and rewrites. It started as an ISPF dialog using ISPF tables as its database. In the late 1980's, the commercial version went from ISPF tables to VSAM files. In 1998, ISPW was completely re-architected and rewritten from VSAM to DB2, C and Java.

Initial users:

Originally, ISPW was written as a programmer's workbench for application developers, so its focus was on developer productivity and helping people use their tools, such as Natural, CSP, QMF, Focus, Expediter, File-Aid and so on. Later, the audit and production control requirements were incorporated, and ISPW became the common teamware tool across all of IT.

Around the same time, ChangeMan and Endevor were being developed for Production Control users to help them migrate application code through the back end of the change cycle to Production.

Initial source and executable components:

Shell Canada had a complex mix of application tools and utilities that were all required to be managed by ISPW. Some of the source code was in standard PDSs, but some source was in Adabas files and some was in VSAM and BDAM files. The executables were to run in batch, Adabas, VSAM and CICS. So from day one, ISPW was already required to checkout, edit, version, compile and promote modules residing in any type of source file (not just PDSs), and to create all types of executables (not just MVS load modules).

Today, everything goes. Applications can have HFS, HTML, Windows, ASCII, binary, and dozens of other languages and files types. Fortunately, the initial requirement for ISPW to handle any flavor of source or executable file is as applicable today as it was back then.

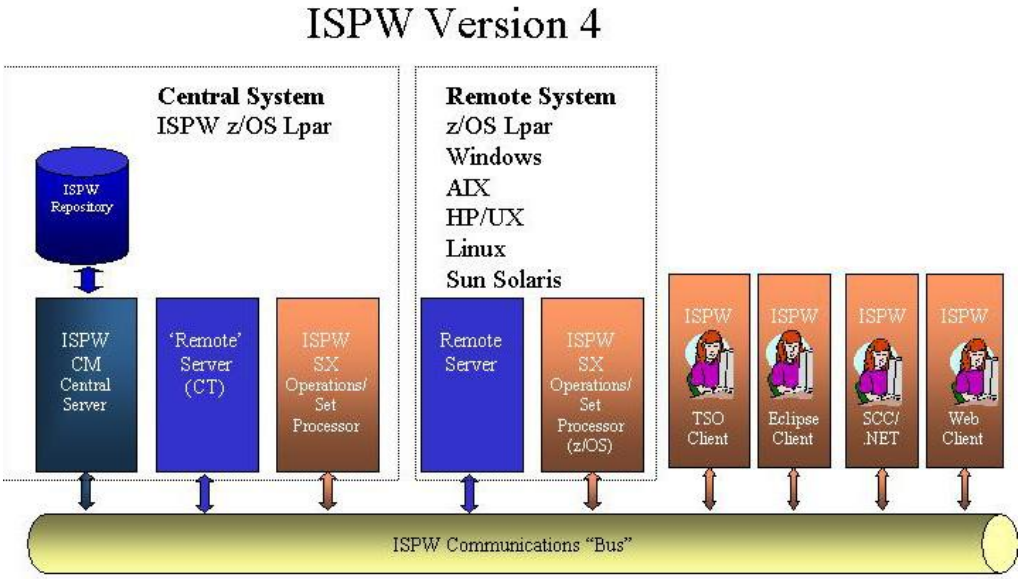
Initial tools and utilities:

Along with code management, ISPW was also required to integrate and invoke tools such as JCL checkers, debuggers, test tools, editors, and so on. Shell had Expediter, File-Aid, JCLCHECK, and a host of other utilities that ISPW integrated so developers would be prompted with the right tool at the right time, and the company would realize a better return on its tools investment.

Today's tools and utilities:

Over 27 years, ISPW has become increasingly more sophisticated, especially in its ability to manage complex DB2, IMS and WebSphere applications, even across distributed platforms.

How ISPW works:



This graphic shows ISPW's modular architecture with its central server components, its remote deployment environments and its user interfaces. ISPW is a single seamless product that controls and manages application development and deployment across both the mainframe and distributed platforms.

ISPW is built on an n-tier architecture structure with several client and server components. The Master (primary) ISPW server provides services on a transaction/application server basis and is z/OS based.

ISPW has three main user interfaces – a TSO 3270 interface, a Web Browser and an Eclipse-based GUI, plus an SCC plug-in. These user interfaces are simply different thin clients which all connect to the same ISPW server. There is only one ISPW product to install, learn and support.

ISPW's control information is in an underlying DB2 database of 75+ DB2 tables. This control/meta data defines the organization's modules, applications, life cycles, approvals, tools, in fact any information about the way Applications are built and managed within the company.

ISPW's Master server accesses DB2 itself, so individual ISPW users do not require direct DB2 access. The ISPW Master communicates with DB2 via CAF. All SQL access is static. ISPW technical support staff administer this control data through a dialog, so there is no need for an ISPW administrator to know DB2.

ISPW stores versions of application source and executables in the ISPW Component Warehouse. These versions may be in any format (text or binary), and any number of versions may be stored for any type of component, at any level. ISPW Versions have internal version numbers, and based-upon version numbers, to assist in parallel development tracking and conflict resolution.

ISPW Releases are composed of specific versions, so it's possible to see that Version 1.2 of a specific module is included in Release 2.7 of a specific application, for example.

More specifically:

1. ISPW is a DB2 application – there is no "hard code" in ISPW. ISPW is a modern well architected repository-based product using DB2 as its store for control and meta data. Its layered architecture has the business logic in the master server close to DB2, and the various user interfaces all interact with it in the same way.

   There is no code in ISPW that says "If this is an MFS screen, do this" or "If this is a DB2 module, do that". All of the processing is entirely table driven. There are about 75 DB2 tables driving ISPW, which is one of the differences between ISPW and some of the other SCM products.

2. ISPW is a Started Task implementation – which needs to run authorized as it performs security checks and both cross-memory and cross-system services. ISPW doesn't just use its Started Tasks for security. They are the backbone of the communications layer between LPARs and even to remote servers, like AIX, Linux and Windows.

3. The code – is in Assembler, C and Java, with some Rexx.

4. Security – is based on SAF, so Sites can have RACF, ACF2 or Top Secret. Security for ISPW operations is mapped to SAF Resource Names, which means all of the work around security is done outside of ISPW, in the security product.

5. The communications layer – the ISPW Started Tasks communicate over an ISPW "bus" that runs over TCP/IP. It encrypts and compresses the traffic to support users working with ISPW through its green screen interface as well as from the ISPW Eclipse GUI.

6. Component types – are 4 alpha-numeric characters, Customer-defined. ISPW is shipped with a pre-populated set of acronyms, such as ASM for Assembler programs, but Customers can modify modules types to whatever makes sense to them.

   For example, one of our banking Customers referred to their Assembler programs as BAL modules, so that became their component acronym for Assembler source.

7. Application codes – are also up to 4 alpha-numeric characters and are the acronym for an application or system, such as the Payroll application, or the Online Banking system.

8. How applications and modules are registered in ISPW – the ISPW administrator simply inputs the file names and types of the source and executable datasets (at Production, QA, etc.) into the repository, and an ISPW batch job then reads through the source files associating modules with applications.

   For example, if copy member ABC is in the Payroll Copylib, ISPW then tags that copy member as belonging to Payroll. The key is Application - module-type – module-name. If a copy member called ABC is also found in a Copylib belonging to the General Ledger application, ISPW can even differentiate between those two modules and knows they are not the same code.

   And most importantly, code doesn't have to be recompiled to be registered in ISPW!

9. How change impacts are determined – at the same time as modules are scanned and registered in ISPW, it also builds up a map of how the application hangs together. By scanning for keywords such as "CALL" and "COPY", ISPW also determines which modules are called or referenced by other modules, so change impacts can be determined.

   ISPW can even handle nested copybooks (i.e. a copybook within a copybook, macros within macros, and Includes within Includes), and even in-house developed copybook syntax/processing. And it can support dynamic call helpers. For example program A has: CALL ZW27 using called-program, parm1, parm2, parm3, … In the working storage of Program A there may be a data name of CALLED-PROGRAM. The value is a max of 8 characters and is the load module name that needs to be dynamically called. For this example lets say the value is "SUBPGM1. What happens is Program A calls program ZW27. ZW27 Loads the value of the first parm (SUBPGM1). ZW27 then calls SUBPGM1 using parm1, parm2, parm3, etc.

   The reason why ISPW can handle these conditions is its parser infrastructure supports User exits (samples are provided) which allow a customer to accommodate any local requirements.

   In summary, ISPW knows which Copybooks are referenced in which Subroutines, which are called by which programs, which are invoked by which PROCs within which batch jobs. And every time a component is updated and promoted, this information is automatically reparsed and refreshed.

10. Versions – both z/OS (source, load, listings, etc.), and open systems components (e.g. java, gif, exe, etc.) are supported. Files are stored in their local file system (e.g. for compiles) as required. An unlimited number of backup versions can be stored.

11. The ISPW Warehouse – is a standard PDS/E based implementation. It's controlled by an ISPW Started Task that manages the space, which proactively avoids x37 space problems.

    It's important to note that both mainframe and distributed source and executables are stored on the mainframe, in the ISPW Warehouse, so there is only one location for all of the company's code assets. As well, the Site's standard backup and DR processes then apply.

12. A "version" is the entire module – ISPW zips the module going into the Warehouse, and unzips it coming out, so it does not "delta" the code as some other products do. This technique of storing the full version works for any type of component (even binary files), and means that the recovery of older versions is instantaneous.

13. Any module may be versioned – text or binary, EBCDIC or ASCII. Every mainframe and distributed module type is supported, including executables like JAR, WAR and EAR files. ISPW knows the module type and does not translate or convert it. The module is simply zipped and stored, unchanged, into the Warehouse.

14. Compile/generate routines – are performed with standard ISPF skels and file tailoring, so there is nothing proprietary in the way compiles are done. Executables can be created for any type of module, even Natural programs, CA-Telon code, third party software, and so on.

15. How compiles/generates work – ISPW ships with a set of sample skeletons that are slightly modified at each new customer site. These skels are mostly symbolic parameters that are resolved as the compile JCL is generated.

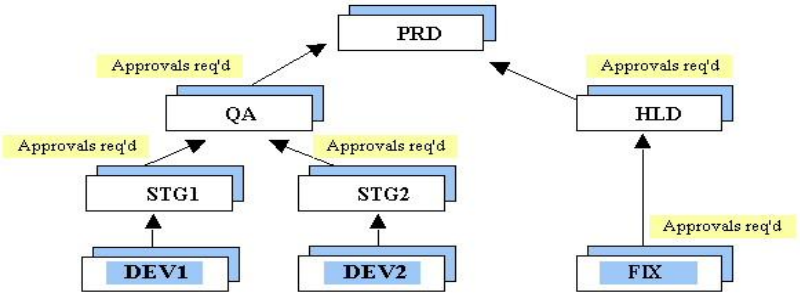
    The compile routine reads each module's specific compile parameters (e.g. compiler version, additional parameters), from the ISPW DB2 repository and builds the JCL on the fly. When complete, it submits the fully built compile batch job to the internal reader. Typically, most ISPW Customers only have 20-30 compile stubs which require almost no maintenance.

    Customers can add variables to the ISPW repository that are retrieved and available during the Compile JCL build time. For example, a Customer-specific variable XXX may be referenced during the compile JCL build time so that skeleton logic can be driven off of this variable.

    In fact, this ISPW-specific extension data (in the DB2 repository) was developed to handle virtually any scenario. And this feature is not as "raw" as the Customer having to add DB2 columns – we provide a layer above that to make it more customer-friendly and panel driven.

16. Life cycle definitions – are simply parameters in the ISPW DB2 repository, and an application can have any number of Paths and Levels.

    Typically, an organization will define a standard life cycle that most applications adopt, though some applications may add or delete Paths and Levels as they require.



    This is the life cycle for the PLAY application on the ISPW Demo system. It has 3 paths, with 3 levels in two of the paths, and 2 levels in the emergency fix path.

    Code is typically checked out of the PRD level, updated in the DEV1, DEV2 or FIX levels, then migrated back up to Production.

    In this example, day to day maintenance may occur in the DEV1-STG1-QA path, a new release of the application may be under development in the DEV2-STG2-QA path, and the third path, FIX-HLD, would typically be used for emergency quick fixes.

17. Libraries are usually sparsely populated – typically, only modules in motion in the change cycle are physically present in the libraries at each level. In other words, the datasets are empty except for modules that are being updated or tested as they migrate up to Production.

    Every application may have its own separate life cycle, i.e. unique paths and levels. A Site would typically set up a standard life cycle that would apply to most of their applications, and then override those definitions for specific applications, on a case by case basis.

18. How concatenation works – the ISPW compile process automatically concatenates libraries in the life cycle, where compiles at the DEV1 level in the graphic above would concatenate DEV1 before STG1 before QA before PRD.

    This information is not "hard coded". It is entirely table-driven from ISPW's internal life cycle map of each application and how the various libraries are concatenated together.

19. How associations work – this interesting ISPW capability is often used to associate or "bring in" other libraries such as common Corporate-wide Copylibs or Link libraries. Again, this is simply done with a parameter in the repository, which are by Application.

20. How code migrates through the change cycle – changes can be promoted (migrated) up to the next level either individually or in groups with the ISPW P (Promote) operation.

    The Promote operation typically moves the source code and regenerates the executables at the next level, if requested. The code at the lower level is typically deleted, though it can be retained, which is as easy as setting an Application- specific Flag to "Y".

21. Installing executables - in ISPW, there are 2 ways to install executables into specific runtime libraries – with the ISPW Promote operation that elevates the source and regenerates the executables, and with the ISPW Deploy operation that simply installs the executables into a target test or production runtime. ISPW is extremely flexible in deploying code anywhere it's required.

22. Extensibility – ISPW's architecture allows it to be easily extended to handle any type of component and even to handle additional custom attributes that may be required. As noted in an earlier point, this ISPW-specific extension data (stored in DB2) was developed to handle virtually any scenario.

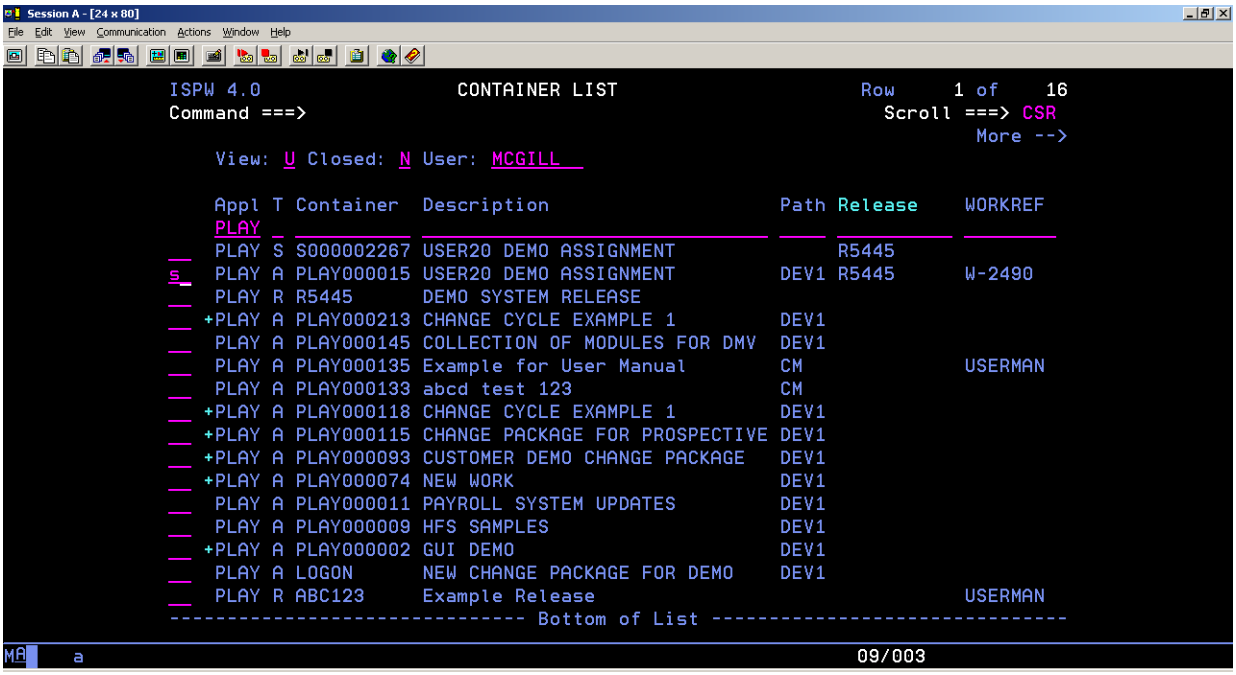
    And this feature is not as "raw" as the Customer having to add DB2 columns – we provide a layer above that to make it more customer-friendly and panel driven.

23. ISPW Change packages – these last few points are the heart of ISPW and the main difference between ISPW and other SCM products.

    ISPW has the concept of "Containers" which are like change package folders. Though other SCM products have change packages and change numbers, these concepts are quite different in ISPW.

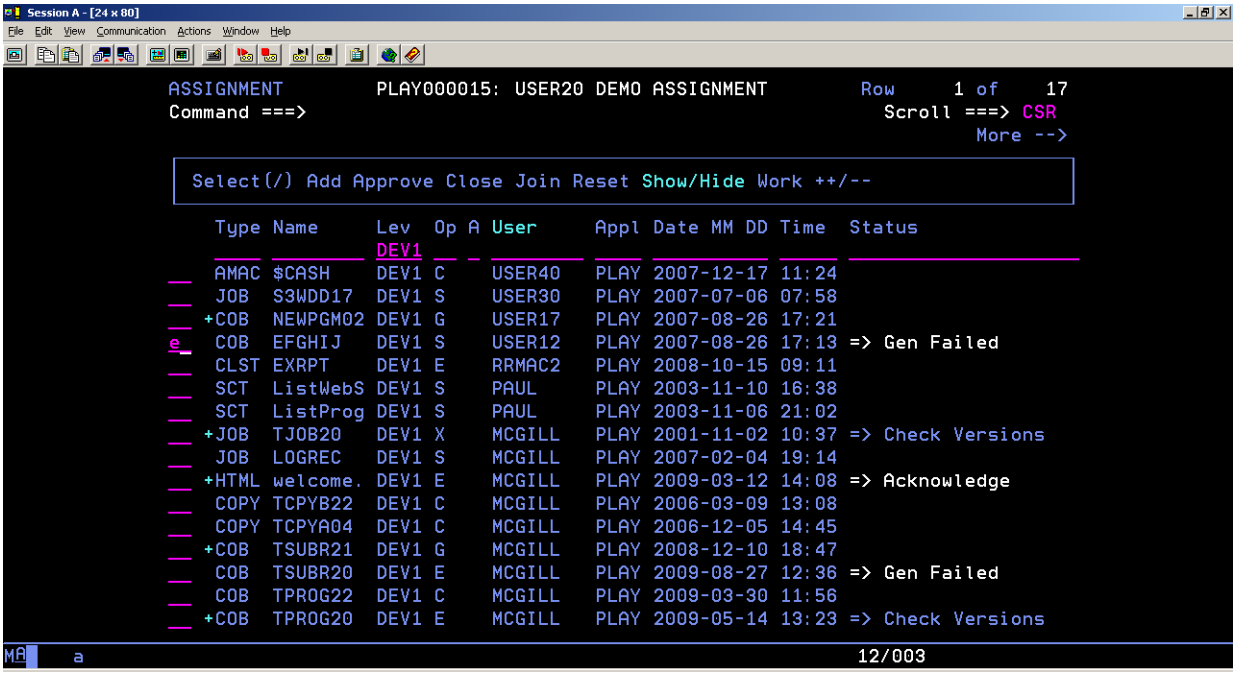
    Below is a screen shot of the 16 "Containers" (change packages) that are associated with user MCGILL. A container is like a folder for a change package, and a change package can pertain to a work request, to resolve a problem ticket, perform some maintenance on an application, and so on.

    In this screen shot, these Containers/change packages all belong to the PLAY application, they each have a brief description and they have optional Release and Workref numbers. The pink text fields are filters on the Container list, and this display is sorted by Release (note the column header is a different color).



Developers do their work in Containers (change package folders), which are typically created at the start of a change process, they're open while work is being done, and they're closed and archived once changes have gone to Production.

This screen shot is the Task List for Container number PLAY000015, filtered by Level and sorted by User.



Containers have "Tasks" (rows) which are basically pointers to the items that make up a change package. Tasks can refer to Cobol programs, JCL, program documentation, project specs, database updates, job scheduling changes, in fact anything that's part of a project.

Working with a module is as easy as typing a command beside the Task to invoke an operation such as Checkout, Edit (shown beside the Cobol program in this screen shot), Generate, Compare, Promote, and so on.

In summary, though other SCM products use change numbers to group components together, an ISPW Container is a different concept. It's like a logical window to the work in a change package that sits "in front of" ISPF. It gives developers a more productive teamware environment to do their jobs done faster, with better visibility and communication, and with less training.

Please see the ISPW demos on the web site for a live demonstration of how this works.

24. How ISPW operations work – all of the variables we've discussed so far are in the ISPW DB2 database, and the ISPW Operations are in the DB2 database as well!

    When the user types an operation, such as E for Edit beside one of the rows in the Task List, ISPW passes the operation, module type and level as keys to the mainline routine which looks up the associated dataset names, file types, technology (ie. Natural, QMF, Telon, Word, etc.), and calls the appropriate sub-routine, which it also reads from a DB2 table. (i.e. Even operations are in the DB2 repository, which means customers can easily add their own site-specific operations.)

    Every operation can even have pre-op and post-op routines. For example, most Customers have change control and problem ticket systems tied in to ISPW. So, a pre-op might check for a valid problem ticket for this module, or maybe an external change control number.

    And that's why customizing ISPW is so easy!

What is ISPW?

http://www.youtube.com/watch?v=i4hrv-ES6mE&list=PL03945E0A91431829&index=1

How ISPW works in a Distributed Environment

http://www.youtube.com/watch?v=8CDJMbmDhy0&list=PL03945E0A91431829

How ISPW works on the Mainframe

http://www.youtube.com/watch?v=3UNYpjRFIz0&list=PL03945E0A91431829

ISPW Demo

http://www.youtube.com/watch?v=zQm-Kp2oS28&list=PL03945E0A91431829

Why are people switching to ISPW?

http://www.youtube.com/watch?v=QyTG9e9Ff2Y&list=PL03945E0A91431829