

Driving Software Quality with Continuous Testing for IBM z/OS Applications

William Alexander

Kimberly Bailey

Suman Gopinath



IBM Z

IBM
®

Redpaper



IBM Redbooks

**Driving Software Quality with Continuous Testing for
IBM z/OS Applications**

November 2022

Note: Before using this information and the product it supports, read the information in “Notices” on page v.

First Edition (November 2022)

This edition applies to z/OS 2.5.

© Copyright International Business Machines Corporation 2022. All rights reserved.

Note to U.S. Government Users Restricted Rights -- Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Contents

Notices	v
Trademarks	vi
Preface	vii
Authors	vii
Now you can become a published author, too!	viii
Comments welcome	viii
Stay connected to IBM Redbooks	viii
Chapter 1. Changing the status quo: Shift Left	1
1.1 Evolution of testing in software development	2
1.2 Shift Left: Testing early and often	3
1.3 Test automation	4
Chapter 2. Applying agile testing and tools to IBM z/OS applications	7
2.1 Examining the levels of testing and choosing the correct tool for the job	8
2.1.1 Unit testing	8
2.1.2 Integration testing	12
2.1.3 Additional types of testing	13
2.1.4 Test data	14
2.2 Testing as an extension of code and modern Source Control Management	14
2.3 Continuous testing	15
Chapter 3. Planning for an implementation	17
3.1 Installing ZUnit	18
3.2 Configuring ZUnit	18
3.2.1 Configuring the host-side components	18
3.2.2 Configuring the client	22
3.3 Installing IBM Z Virtual Test Platform	28
3.4 Configuring the IBM Z Virtual Test Platform	29
Chapter 4. A sample scenario	35
4.1 Testing and understanding the transaction flow of the application with the IBM Z Virtual Test Platform	36
4.2 Testing single programs by using unit testing	39
4.2.1 Creating a ZUnit test	39
4.2.2 Making application changes	44
4.2.3 Testing the program before commitment	44
4.2.4 Integration testing	48
Chapter 5. Best practices and recommendations	53
5.1 Process and cultural recommendations while adopting test automation	54
5.2 General best practices	55
5.3 Summary	55
Abbreviations and acronyms	57

Notices

This information was developed for products and services offered in the US. This material might be available from IBM in other languages. However, you may be required to own a copy of the product or product version in that language in order to access it.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not grant you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing, IBM Corporation, North Castle Drive, MD-NC119, Armonk, NY 10504-1785, US

INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some jurisdictions do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM websites are provided for convenience only and do not in any manner serve as an endorsement of those websites. The materials at those websites are not part of the materials for this IBM product and use of those websites is at your own risk.

IBM may use or distribute any of the information you provide in any way it believes appropriate without incurring any obligation to you.

The performance data and client examples cited are presented for illustrative purposes only. Actual performance results may vary depending on specific configurations and operating conditions.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

Statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to actual people or business enterprises is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. The sample programs are provided "AS IS", without warranty of any kind. IBM shall not be liable for any damages arising out of your use of the sample programs.

Trademarks

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corporation, registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the web at "Copyright and trademark information" at <http://www.ibm.com/legal/copytrade.shtml>

The following terms are trademarks or registered trademarks of International Business Machines Corporation, and might also be trademarks or registered trademarks in other countries.

CICS®
CICSplex®
Db2®
IBM®

IBM Z®
Language Environment®
Rational®
Redbooks®

Redbooks (logo) ®
VTAM®
z/OS®

The following terms are trademarks of other companies:

Evolution, are trademarks or registered trademarks of Kenexa, an IBM Company.

Windows, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

Java, and all Java-based trademarks and logos are trademarks or registered trademarks of Oracle and/or its affiliates.

Other company, product, or service names may be trademarks or service marks of others.

Preface

This IBM® Redbooks® publication provides IBM z/OS® development and test organizations with the information to develop and establish agile testing practices for their z/OS applications. It is assumed that readers know about software development, quality assurance, agile development processes, and DevOps.

You might think that applying agile testing to z/OS applications is not practical or efficient due to a lack of good tools. In this publication, we hope to dispel that myth by sharing how developers and testers can apply agile testing practices to their z/OS applications and the tools that can help to accelerate their processes. The intended outcome is for z/OS development organizations to compare how they are testing today and determine whether there is an opportunity to transform testing practices and apply new tools to improve quality and gain efficiencies.

The target audiences for this publication are organizations that develop z/OS applications, specifically development and quality assurance managers and team leaders that recognize the need for their teams to improve quality and gain efficiency.

Authors

This paper was produced by a team of specialists from around the world working at IBM Redbooks, Poughkeepsie Center.

William Alexander has over 25 years of experience in mainframe application development. Before joining IBM, Bill developed and supported COBOL batch and IBM CICS® applications for MetLife and Bank of America. Since joining IBM, he has been involved in the design and development of various middleware and tool products. Bill is a Senior Technical Staff Member with the IBM Z DevSecOps team who focuses primarily on IBM Developer for z/OS (IDz). Bill holds multiple patents and often presents at various conferences regarding methods to increase the productivity of mainframe application developers.

Kimberly Bailey has over 30 years experience with IBM, primarily working on IBM Z®. She has held roles across the spectrum of development, business development, and product management. At the time of writing, she is the product manager for Shift Left Test Tools in the IBM Z DevOps organization.

Suman Gopinath is an IBM Senior Technical Staff Member, architect, and CTO with the IBM Z DevSecOps team. She has worked in the mainframe application development space for over 15 years. She has worked on DevOps and automation for the last decade. Suman helps customers with their DevOps journey in the mainframe space and helps them build solutions to help create an end-to end-pipeline. She is the architect for testing solutions on IBM Z.

Thanks to the following people for their contributions to this project:

Lydia Parziale and Wade Wallace
IBM Redbooks

Sherri Hanna and Shalani Mohan
IBM

Now you can become a published author, too!

Here's an opportunity to spotlight your skills, grow your career, and become a published author—all at the same time! Join an IBM Redbooks residency project and help write a book in your area of expertise, while honing your experience using leading-edge technologies. Your efforts will help to increase product acceptance and customer satisfaction, as you expand your network of technical contacts and relationships. Residencies run from two to six weeks in length, and you can participate either in person or as a remote resident working from your home base.

Find out more about the residency program, browse the residency index, and apply online at:
ibm.com/redbooks/residencies.html

Comments welcome

Your comments are important to us!

We want our papers to be as helpful as possible. Send us your comments about this paper or other IBM Redbooks publications in one of the following ways:

- ▶ Use the online **Contact us** review Redbooks form found at:
ibm.com/redbooks
- ▶ Send your comments in an email to:
redbooks@us.ibm.com
- ▶ Mail your comments to:
IBM Corporation, IBM Redbooks
Dept. HYTD Mail Station P099
2455 South Road
Poughkeepsie, NY 12601-5400

Stay connected to IBM Redbooks

- ▶ Find us on LinkedIn:
<http://www.linkedin.com/groups?home=&gid=2130806>
- ▶ Explore new Redbooks publications, residencies, and workshops with the IBM Redbooks weekly newsletter:
<https://www.redbooks.ibm.com/Redbooks.nsf/subscribe?OpenForm>
- ▶ Stay current on recent Redbooks publications with RSS Feeds:
<http://www.redbooks.ibm.com/rss.html>



Changing the status quo: Shift Left

In the last few years, we have become accustomed to reading about companies in the news who have had software failures that are serious enough to impact markets. There are many reasons for these failures, but poor software quality is certainly one. People who work in software development know that ensuring quality in complex industry applications is difficult and that poor quality can lead to failures and security holes. When these issues become headlines, all of these people can empathize.

High-profile incidents are not surprising because organizations are driven to deliver products to market faster at the lowest cost with quality. To meet that objective, many organizations adopt software development practices like *agile*, which is a method for faster delivery where incremental functional deliveries enable testers and other stakeholders to provide rapid feedback on quality and usability. A DevOps pipeline of continuous integration and continuous delivery (CI/CD) capabilities has automation at its foundation. Agile and CI/CD are not new, and most organizations have established them to some degree, so this chapter is not intended to be a lesson. However, in discussions with large enterprises running on z/OS, it is clear that although many have some form of test automation, they still lag in adopting agile testing practices like Shift Left and continuous testing, which are key to businesses fully meeting their objectives.

In discussions with z/OS clients, many state challenges with processes that are delaying continuous testing. The top reasons that are cited are the following ones:

- ▶ Shared testing environments, which cause delays.
- ▶ Lack of test automation frameworks and tools that work across Hybrid z/OS applications.
- ▶ Lack of tools for Unit and early Integration Testing for COBOL, PL/I, and assembly language applications that support Shift Left, such as autovalidation of test and record and replay capabilities.
- ▶ Lack of skills and tools to fully develop good, automated test cases that are end-to-end for z/OS hybrid applications. It is not easy for new testers to build their own programs to interface with batch, IBM CICS, and TN3270 applications and middleware.

Many of these challenges can be addressed today. There are tools that can help accelerate testing for IBM z/OS applications.

Do not let myths about testing z/OS applications prevent you from transforming testing in your organization that can greatly improve your business objectives and the quality of your deliverables. In this chapter, we describe the evolution of testing in software development and the concept of Shift Left Testing. Additionally, we describe test automation.

1.1 Evolution of testing in software development

Testing is an important phase of software development. In the waterfall model, it is a separate activity after the coding or “development” phase, so it is a silo by itself. With the advent and popularity of the agile software development model in the 2000s, testing became something that needed to be more close-knit to development and keep up with incremental agile iterations. Faster, smaller increments also meant that tests needed to be automated and efficient. With digital transformation and technology disruptors in the market, agile development and test automation became necessary.

Figure 1-1 shows the typical test pyramid.

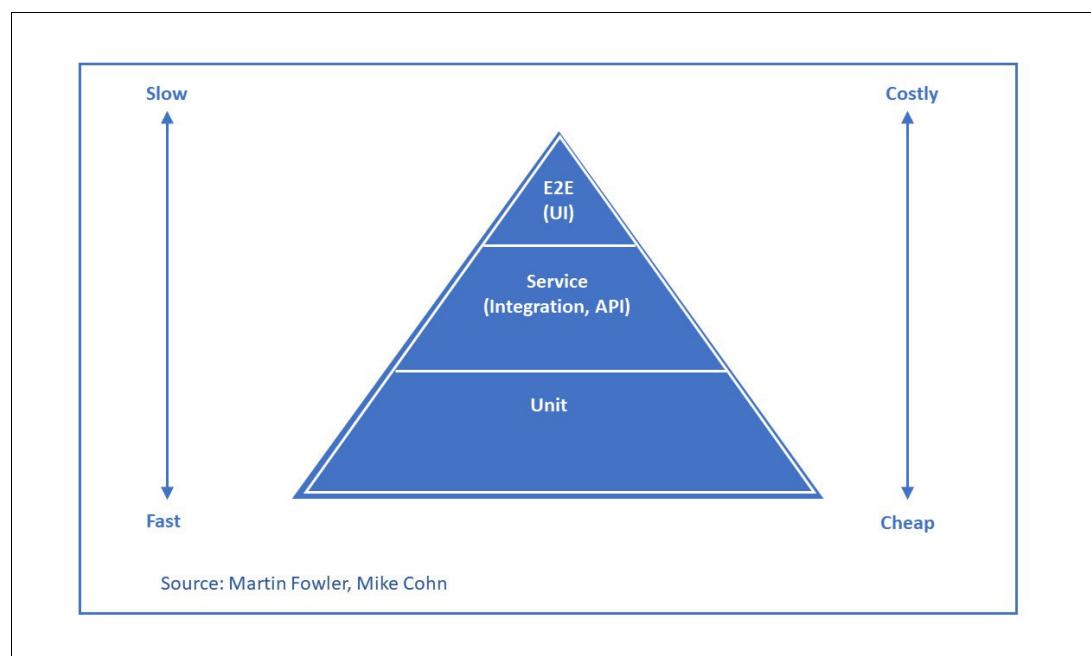


Figure 1-1 The test pyramid

Two categories of testing are “black box” and “white box” testing, which specifically relate to the level of knowledge that the tester has about the application that is being tested, and the levels of testing that they perform. The bulk of testing in enterprise z/OS applications focuses on functional testing or ‘black box’ testing (shown in Figure 1-2 on page 3), where the internal structure or implementation of the components or functions that are tested are unknown to the tester. Functional testing refers to testing the way that a screen is supposed to behave if the correct validations are presented to the user or if the correct values are retrieved into the screen or messaging system. This kind of testing is essential to any software, and it helps ensure that the software functions correctly when given the correct input.

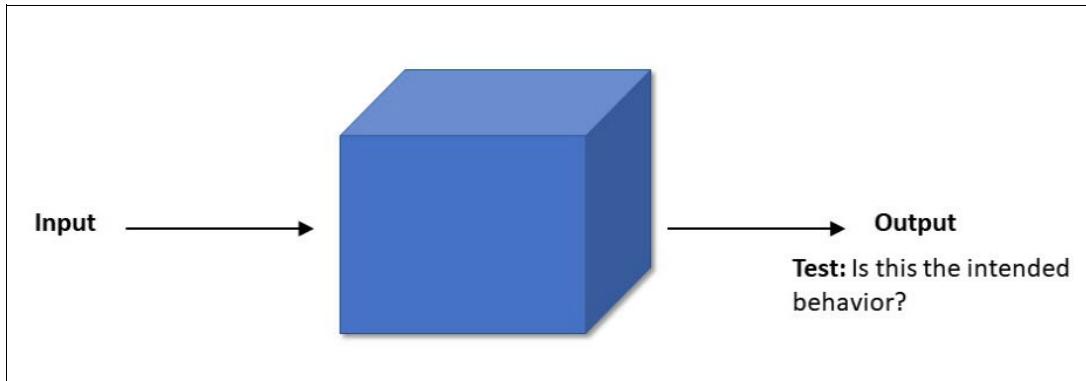


Figure 1-2 Black box testing

Black box testing depends less on the technology behind the implementation of the functions. The testing of the implementation can be performed by a different team that is independent from the developers. Because the tests are done from a user point of view, the test cases can be designed when the specifications or requirements are complete. Black box testing is useful for capturing security vulnerabilities that are associated with a designed user flow, for example, invalid configurations, validation errors, and other runtime issues. The typical test automation tools that are available for black box testing falls short in its ability to validate every part of a z/OS application, such as the values in an IBM Db2® table or in a sequential file.

Agile development tests much earlier in the software cycle. Detecting a defect in the functions of the single program under test and testing boundary conditions before code is checked-in means that all those defects that are otherwise caught in integration testing in a shared environment are fixed much earlier. The next stage of testing can concentrate on detailed functional testing and performance testing and be free from misdirection due to overlooked coding defects. In this era where IT drives businesses, z/OS applications must provide this same kind of flexibility.

1.2 Shift Left: Testing early and often

The concept of Shift Left Testing is about testing earlier in the development cycle, but it also is a paradigm shift in how test and quality should be inherent in each stage of the development cycle.

The idea of unit testing is not unknown to open distributed systems. However, for enterprise z/OS applications, the idea of unit testing has various connotations. The tests typically are not reusable but performed in an *ad hoc* manner that is tied to the whims of the individual developer. To break these practices and embrace the concept of testing the smallest unit of code, testing beyond the functional use case, and then rerunning these tests automatically after each code change requires a process change. This change requires a cultural shift that can be enforced only by the value that it brings.

In this aspect, the granularity of what is tested becomes important. In the early test cycle, unit tests are created by the developer are one more phase before full functional integration testing, and they are tested in increasing granularity. This kind of testing with increasing scope enables an application to be continuously tested without depending on all its interfaces.

For example, a COBOL program can be independently tested, and this testing can be extended beyond the single program, for example, to test integrations with other related programs that comprise the application to uncover defects like interface issues or copybook mismatches before the program is checked in. These tests must be repeated anytime a program changes. The value of this testing is its ability to identify defects and issues in the early development sprints before code is fully merged with other programs and applications to perform the full end-to-end test.

Knowing that each developer has fully ran tests for their parts of the application helps build better trust between agile teams because code that is now merged to form a feature has more quality. The next levels of testing can concentrate on functional tests and not be concerned about incorrect module-level interfaces.

Code must be tested much earlier in the development cycle instead of being sent to the test team or the next stage. Figure 1-3 shows that the unit test and application integration test occur early in development before code is deployed versus after the code is deployed for full integration testing.

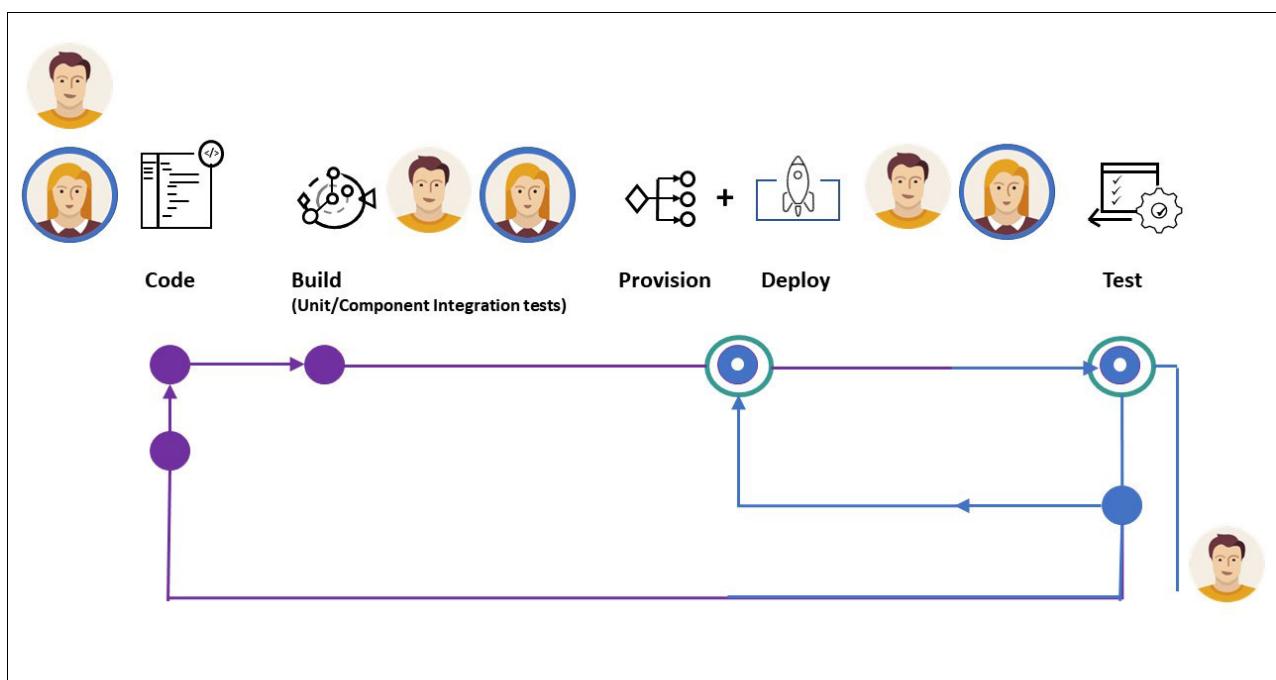


Figure 1-3 DevOps pipeline

1.3 Test automation

Automation is one of the foundational pillars on which the practice of DevOps is built and it should be applied to all phases of testing. Automated tests are developed so that they can be automatically rerun whenever code is changed to test those changes without requiring the test to be performed manually. Checks are built in to the tool to verify that the test completed successfully or it failed based on some expected result. This approach removes the need for individuals to do this task manually. Automated tests are self-validating and have the properties of reusability and repeatability, which drastically cuts the time that it takes to deliver software because it eliminates all the manual work that is involved in setting up tests and verifying the results. The surplus productivity can be used to create and run a wider range of tests that are not tested due to a lack of time and resources.

Development organizations must commit and invest to build an automated testing practice. Most organizations have existing applications, so there is an upfront cost to build tests, and as new code is developed, new tests must be created and existing tests updated. Like application development, automated test cases must be continually updated and maintained.

Being able to automate every stage of your testing so that the tests are repeatable and run on demand is critical to becoming truly agile. An increase in parallel development brings an increased need to include test automation in the enterprise pipeline, whether you are dealing with developer branches, feature branches, or release branches. This approach can help ensure that the quality of the IT application is maintained with the agility in delivery.



Applying agile testing and tools to IBM z/OS applications

As described in Chapter 1, “Changing the status quo: Shift Left” on page 1, using automated testing to ensure that z/OS applications behave and perform as expected reduces the time that is required to deliver applications to users. Thousands of tests can be run programmatically in the amount of time that it takes humans to manually perform a set of testing instructions. In addition, test automation frees humans from the tedious tasks of repeatedly running the same test instructions each time that coding changes are made. This approach provides developers and testers with more time to innovate and focus on activities such as writing more tests, performing usability tests, or exploring “what if” scenarios that otherwise might be ignored.

2.1 Examining the levels of testing and choosing the correct tool for the job

Many studies have been conducted and books have been written about whether it is better for humans to be specialists or generalists. A specialist has deep expertise in a subject matter and can recognize patterns in their specific area to perform above average feats. A generalist has broad knowledge of many areas and can see relationships across different areas so that they can offer insights based on past experiences. When it comes to software testing, most people prefer to use specific tools that specialize in each different layer of testing. Specialized tools have the features that are necessary to ensure a high level of application quality.

As we approach automation for white box testing, which involves testing the nuts and bolts within the program, its boundary conditions, and the way it handles defects, the tools must be more specialized and language-specific. In contrast, a tool that helps in larger functional tests must be designed to support the generalist.

The various testing phases that an application goes through can be broadly classified as follows:

Unit test	Tests that are created and maintained by developers and test a unit of code. For procedural languages like COBOL, a unit of code might be one program.
System test	Tests that verify various aspects of the system. These tests can be <i>functional tests</i> that verify a function; <i>integration tests</i> that verify the integration between programs, components, and applications; and <i>performance tests</i> that test nonfunctional requirements.
User acceptance test	The acceptance tests are run by stakeholders to verify that the change or feature is acceptable to go live. Although automation is a must in all stages of testing, there should be at least a few manual usability tests when testing for feature acceptance.

This paper primarily delves into unit and system integration tests and how these tests can be performed much earlier in a development lifecycle.

2.1.1 Unit testing

A *unit* is the smallest testable part of software, that is, an individual unit of source code, like a single source file. In terms of z/OS application development, the smallest piece of code that can logically be isolated for unit testing is a *program*.

There are two questions that arise when talking about unit tests:

1. What is required to unit test?
2. What is the value that this testing brings?

Unit tests are small, fast, repeatable, and isolated. Isolation in the unit test helps confirm the functions of the unit and that the failure or success lies within the unit and not from external factors. This isolation is implemented by using methods or components that behave like the original production component. In testing vocabulary, these items are known *test doubles*.

Stubs are one implementation of a test double. The stub retrieves predefined data, for example, the data from a **SELECT** statement in a COBOL program instead of invoking Db2 and issuing the statement.

Another implementation of a double is a *mock*. A mock also does not invoke the production component, but instead confirms that a component is invoked with the correct parameters. For example, a mock can be used to determine whether a call to an abend routine due to an invalid SQLCODE is being made without invoking the abend routine and verify that the call is made with the correct error message.

Figure 2-1 shows unit testing with a stub and mock implementation.

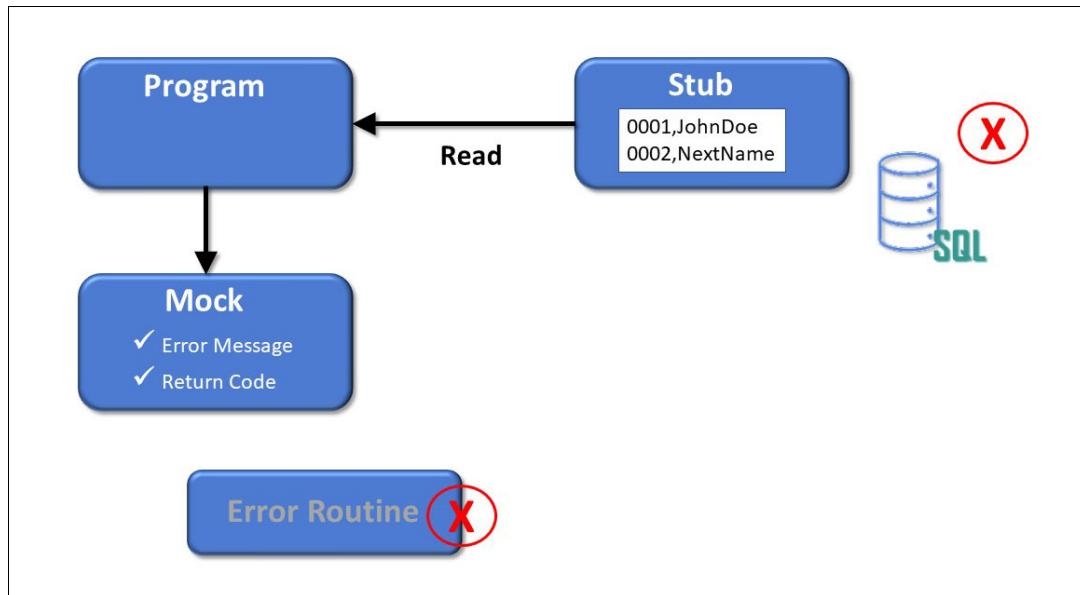


Figure 2-1 Stub and mock

A unit testing tool or framework must be able to perform small, fast, repeatable, and isolated tests. A developer must be able to programmatically determine whether the test passed or failed by specifying the expected values in the test. In addition, *isolated* implies that it should be possible to run the tests with all the interfaces stubbed out or replaced by mocks.

Whenever a developer creates a program or changes an existing program, they also should write a unit test to ensure that their new code behaves as expected before it is integrated with the rest of the application. The idea of unit testing involves writing code to test code. A unit testing framework can help developers with creating such test cases and running these tests in a repeatable manner. The framework can register a test as a unit and associate it with the input and a number of assertions that automatically determine whether the test is a pass or fail. Although the code that is tested eventually is part of the production code, the unit test by itself is not.

Figure 2-2 provides an overview of an approach for unit testing.

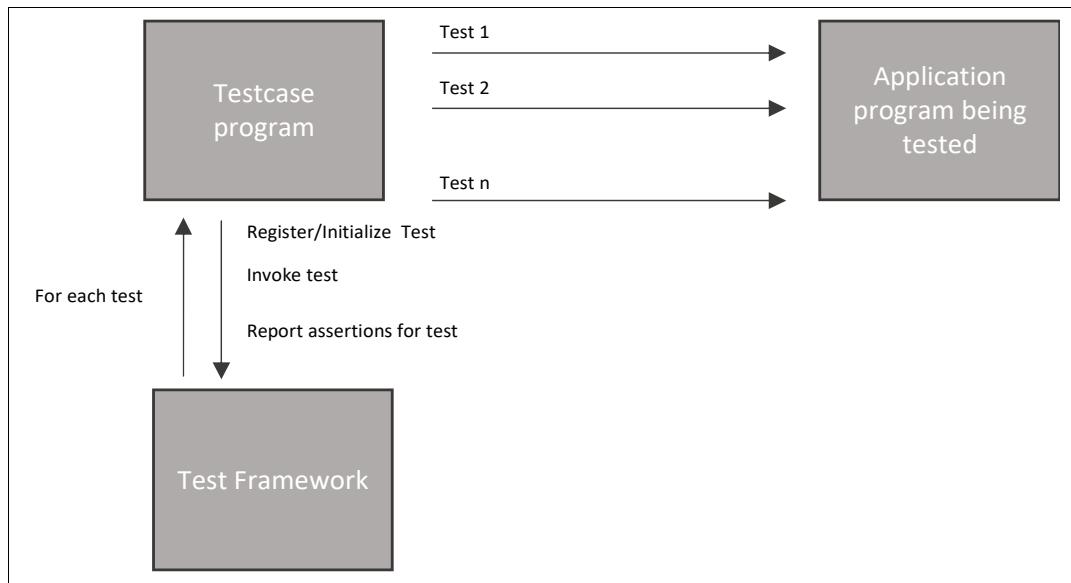


Figure 2-2 A test framework provides a consistent approach to unit testing

The IBM z/OS Automated Unit Testing Framework (ZUnit) is a tool that is a part of the IBM Eclipse Integrated Development Environment (IDE) - IBM Developer for z/OS (IDz), which developers can use to perform unit tests for Enterprise COBOL and PL/I for z/OS applications. ZUnit consists of two main components:

1. Eclipse plug-ins for IDz, as shown in Figure 2-3, is the most popular IDE for z/OS application development.
2. IBM Dynamic Test Runner (DTR), which is a z/OS based test runner that runs the test cases.

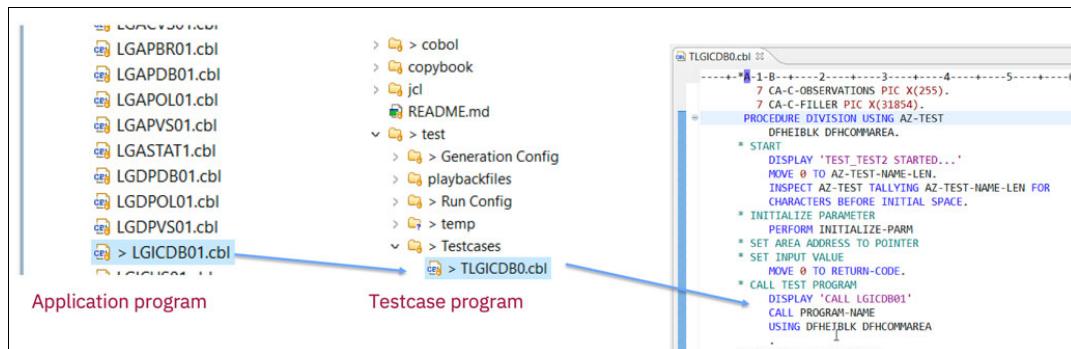


Figure 2-3 ZUnit test case in IDz

IDz modernizes the z/OS application development process for new applications and maintaining applications. Application developers can create and maintain COBOL, PL/I, IBM High Level Assembler (HLASM), Java, and C/C++ z/OS programs by using the IDz Eclipse-based IDE. IDz simplifies coding, debugging, interaction with source control managers, unit testing, obtaining code coverage analysis, and more. The ZUnit tools within IDz consists of a wizard that helps users with creating or editing test cases; a test case editor where users provide the input data to be used for each test and the data that is expected to result from running each test; and a test case generator that accelerates test creation by eliminating the requirement that developers write test code themselves.

The DTR is a z/OS batch application that runs the test case, which is composed of the application code and input data. DTR compares the actual test result with the specified expected result to determine whether a test passed or failed.

The added value that is provided by the DTR is that it can run IBM CICS, IMS, and Db2 tests without the original middleware present and available. For unit testing, this approach allows z/OS application programs to be run in batch in a repeatable manner without having to reset the middleware test environment before each test.

The developer can create a unit test with the ZUnit test case editor in IDz by entering values for the input and specifying the output values to be asserted. However, ZUnit also allows a developer to create a test from a recording of the program. When a developer clicks a record from the IDz editor, ZUnit invokes a recording service that invokes the DTR Collection component. When the recording is stopped, the ZUnit import service imports the data into the editor, and converts the recording and maps it into the I/O data structures and variables that are used within the program. While importing the data, the user may filter the recorded values. A COBOL test case is generated from the input and output values.

Figure 2-4 shows a ZUnit test case that internally consists of various components:

1. The recorded file
2. The run configuration
3. The COBOL or PL/I test case
4. The ZUnit generation configuration

Each of these four artifacts can be stored in a modern Source Control Management (SCM) application like Git.

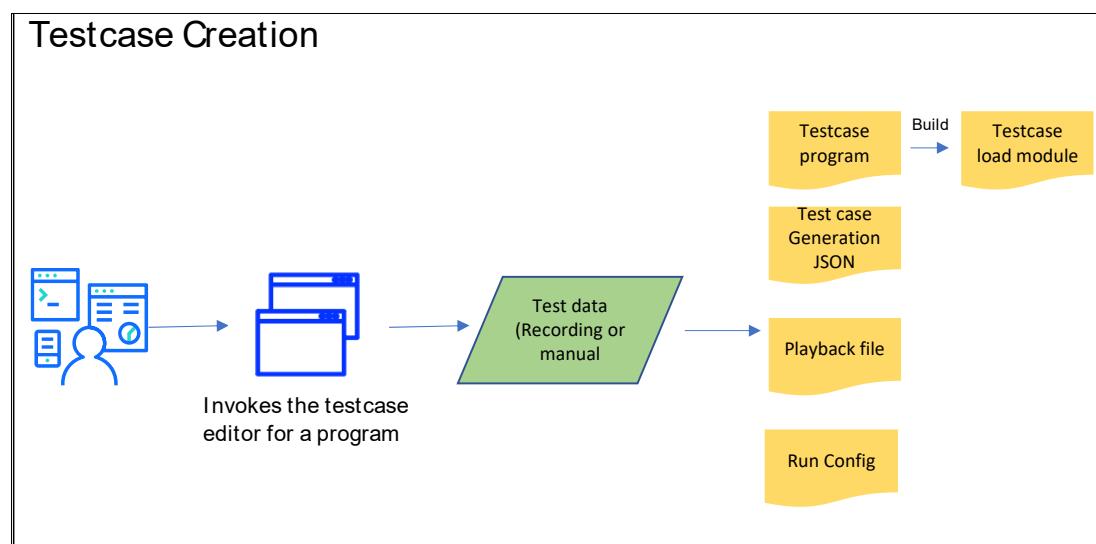


Figure 2-4 Creating a test with ZUnit

2.1.2 Integration testing

An *integration test* is the next level of testing after unit testing. Its purpose is to confirm that the new or modified program works as expected when combined with other programs in the application. Integration testing can uncover bugs in the communication between programs. Just because two programs each successfully pass their individual unit tests does not mean that they will pass an integration test. For example, if there is poor collaboration between the developers working on each of the programs, then mistakes might be made in the interface that is used to communicate between the two programs. This type of failure is what integration testing is trying to uncover. Integration testing can be extended to test the integration between programs, transactions, or the other technical interfaces to an application.

The IBM Z Virtual Test Platform (ZVTP) is a test tool to perform early application integration testing on z/OS applications. It supports testing COBOL and PL/I applications.

ZVTP is composed of two primary components:

1. A web UI for interfacing with the tool to perform key tasks.
2. The DTR, which runs on z/OS and provides capabilities for developers and testers to perform transactional and batch testing without needing the original z/OS middleware.

The DTR supports both ZUnit and ZVTP. An example of integration testing is using the ZVTP to record a CICS transaction and the COBOL programs that are running in CICS for an application. The recorded interactions between these programs are stored in a playback file. This file can be used to replay the transaction and the programs under test in batch by submitting JCL without needing a running CICS environment. The focus of the ZVTP integration test is the interfaces between the programs and not the interaction with the external system.

ZVTP intercepts calls that are made by application programs to various subsystems or other programs and records details about those calls. After this data is recorded, those same application programs can be rerun in batch without needing the original test environment, and feed back the values from the recorded data. This process allows the user to automate the testing process of online transactions and batch programs, whether they are testing a single program, hundreds of programs, or thousands of transactions. Additionally, the application programs can be modified with required changes and then rerun to help ensure that they perform without adverse effects. Finally, subsystem responses to programs can be altered during collection by using user commands, or during replay by using exit points. This process allows users to record and replay the paths that programs do not normally take without having to make any program changes.

Figure 2-5 on page 13 shows the ZVTP two-part process for testing.

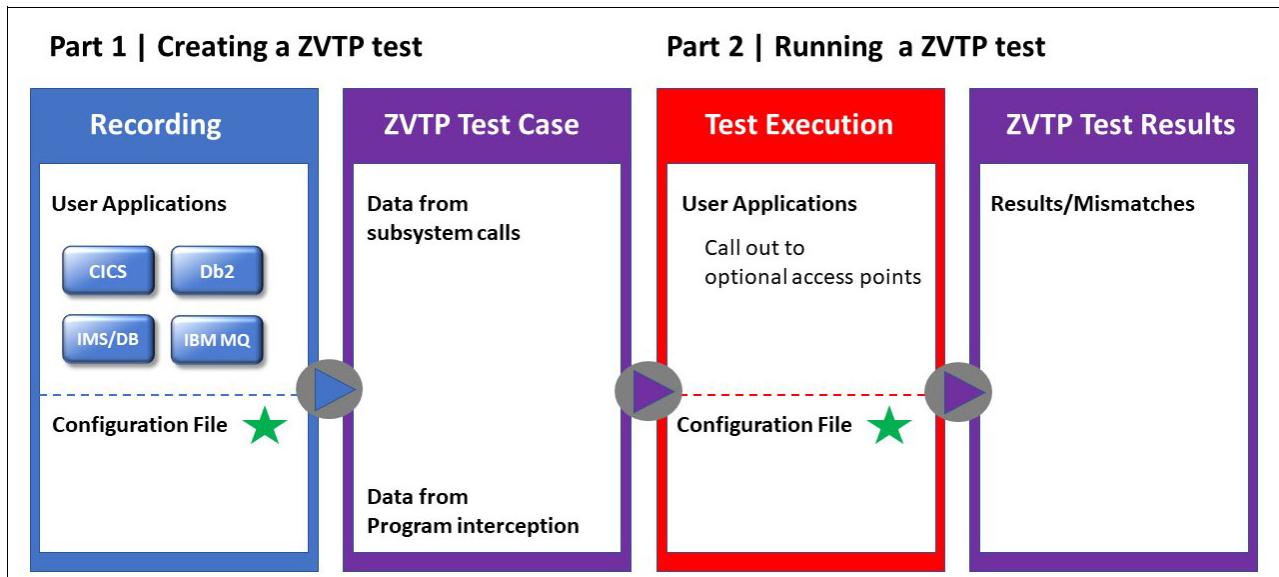


Figure 2-5 ZVTP two-part process for testing

Integration testing can be performed at various levels. Starting with ZVTP at a program integration level, this testing can be expanded to one application with certain interfaces that are virtualized with products like IBM Rational® Test Workbench.

Open-source test frameworks like Galasa allow for automating these integration tests by interfacing with z/OS and APIs, and by using test tools like Selenium.

Early tests help a developer to confirm that the changes that are made at a unit level and program-to-program integration level are correct. These tests also help to ensure that various boundary conditions are tested. However, these tests do not eliminate the need for full integration tests after deployment that confirms that the values are updated correctly in the database. There are different ways to automate these tests, such as by using IBM Rational Test Workbench or the Galasa test automation framework.

2.1.3 Additional types of testing

Other types of testing that organizations perform are as follows:

Performance testing This type of testing ensures that the application can perform well with a high volume (load) workload over time. The application behavior is monitored to ensure stability. Common checks ensure scalability, good response time, no memory leaks, and the ability to handle temporary resource shortages. Testers use several homegrown and standard tools to drive workload and monitor the results.

API testing Core business-critical applications and data that is hosted in z/OS can be exposed as APIs for consumption by new cloud-native application logic, such as mobile or cognitive applications. With no new code changes to the z/OS applications, modern solutions can be delivered through new channels to provide enhanced interactions with cloud-based solutions. IBM z/OS Connect supports the development of APIs that are based on the OpenAPI Specification from core applications on IBM Z, and a robust, comprehensive runtime environment for exposing APIs.

The specification is the contract that binds the communication between the requesting application and the responding application. Therefore, it requires testing when application changes are committed.

Rational Test Workbench includes a component that is named Rational Integration Tester that specializes in service and API testing.

Not all z/OS applications have modern user interfaces. Some older terminal-based applications might contain presentation logic for 3270 terminals. Rational Test Workbench includes another component that is named Rational Functional Tester Extension for terminal-based applications, which can be used to create automated test scripts. It provides a rich set of capabilities to verify host attributes, host field attributes, and screen flow through a host application. Rational Functional Tester Extension uses terminal verification points and properties, and synchronization code to identify the readiness of a terminal for user input.

2.1.4 Test data

The different stages of testing warrant different kinds of test data. For example, the ideal test data for unit tests is fabricated data, that is, data that addresses negative conditions and boundary conditions so that the code is tested in depth for the correct checks. IBM Optim Test Data Fabrication has options to help with this task.

However, end-to-end integration tests need data that is synchronized across various application data stores and databases to perform a full functional flow of data across these applications. This data can be a copy of the production data, but must be sanitized consistently across the enterprise. IBM and IBM Business Partner products provide various options to perform this level of data masking and sanitizing. Investing in a good test data management solution also helps ensure that there is a way for teams to get the correct data in newly provisioned test environments.

Lastly, it is useful to have certain data to perform production level tests. Restricted release of features to certain sets of customers or fictional store codes and orders are all ways to perform a quick production test and validate features. [Enterprise Bug Busting](#) talks in detail about how customers use live data to perform final tests.

2.2 Testing as an extension of code and modern Source Control Management

Traditionally, z/OS application development is performed by using a library manager solution for version control of source code. In these solutions, a single developer checks out the files that they were working on, which locks them while modifications are made. Naturally, this behavior leads to a series of change requests and limited experimentation.

In comparison, modern SCM solutions promote both experimentation, which often leads to innovation, and full parallel development, which accelerates the delivery of change requests. Modern SCMs support multiple branches, allowing teams to isolate several change requests at once, such as test fixes, new features, and maintenance of the current release.

Git is the *de facto* standard SCM for distributed software development. There are many Git vendors, such as GitLab, GitHub, and Bitbucket that have built solutions with their own added value features to the base Git version control system. Universities and open-source projects often use Git for managing source code.

Several enterprises have gone through successful migrations to a modern SCM from traditional library managers that they were previously paying high prices to use for many years. Some of these enterprises had hundreds of thousands of elements in their old library manager. These SCM migrations provided future cost savings; removed vendor lock-in, and allowed for the usage of common tools regardless of platform, shared skills across teams, flexibility in resource assignments; and made the development experience attractive for new hires.

Shift Left Testing requires developers to write and run unit test cases as part of their development activities. A best practice is to store both the application source code and the unit test cases in the same repository, which eliminates any confusion developers might experience when they are required to search for unit tests in another location. When the code that is used for verification is stored alongside the application code itself, it is easy for developers to understand which tests are associated with each program. This best practice also allows developers to quickly maintain both the application source code and the unit test cases as part of one branch or change request, and simplifies review processes.

2.3 Continuous testing

Software development teams are under pressure to deliver application changes in less time and improve quality. The only way to accomplish these goals is through continuous testing and integration into the continuous integration and continuous delivery (CI/CD) pipeline. A CI/CD pipeline can eliminate bottlenecks and manual processes through automation of software builds, analysis, tests, and deployments. Automation is a fundamental component for any CI/CD pipeline.

Today, with continuous integration, each developer integrates their deliverables more frequently than the traditional z/OS application development processes of the past. Pipeline automation is used to help developers by having a build process that compiles the code and provides more feedback by using techniques such as static code analysis and automated testing to identify issues earlier when they are easier to fix. This early identification also helps avoid problems at the final merge when preparing for the release, which results in higher-quality software and more predictable delivery schedules.

There are many choices for continuous integration tools, sometimes called *pipeline orchestrators* or *coordinators* because they orchestrate a sequence of steps to automatically run tasks that were previously done manually. Jenkins is possibly the most well-known automation server, but for enterprises that are interested in a comprehensive platform for DevOps, GitLab Ultimate for IBM z/OS is an all-in-one solution. Besides being a pipeline orchestrator, it is also a Git repository for controlling source code versions, and an issue tracking tool for managing and planning work.

GitLab Ultimate for IBM z/OS also includes a component that is named IBM Dependency Based Build (DBB). DBB provides an intelligent build system with APIs to perform integration with a pipeline. DBB can identify programs that must be rebuilt due to a modified dependency source file. For example, DBB can run ZUnit tests as part of a CI/CD pipeline by providing a ZUnit test dependency scanner, which automatically creates dependency relationships between the z/OS application source program and the test case program, and triggers the unit tests that were created for program verification.

Additionally, open source test frameworks like Galasa (shown in Figure 2-6) allow for automating integration tests and other levels of testing by interfacing with z/OS and APIs, and by using test tools like Selenium.

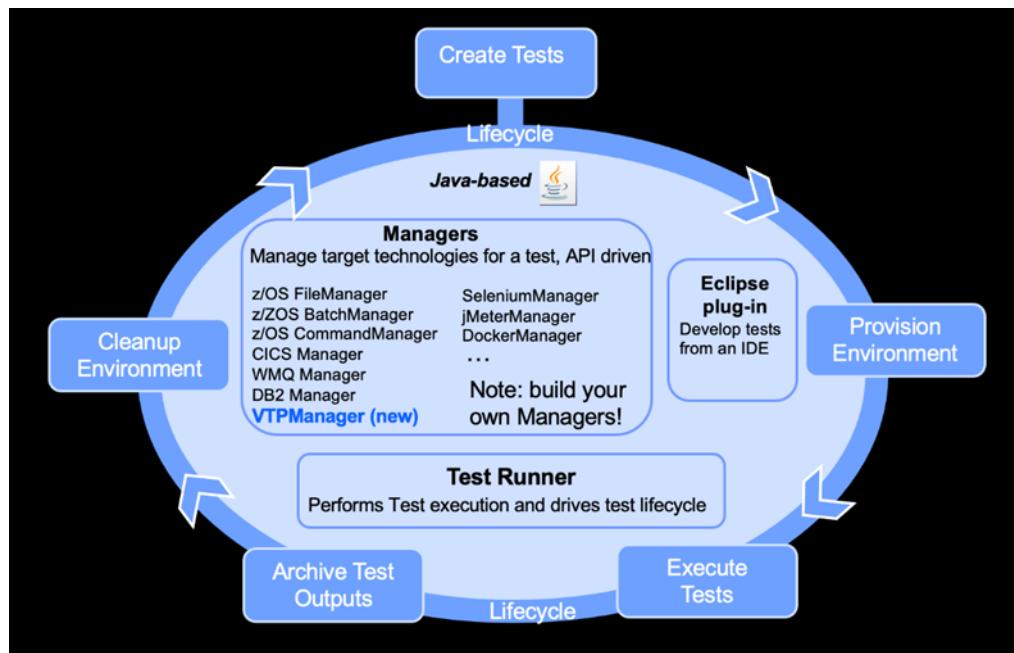


Figure 2-6 Galasa framework

Galasa is a Java based test framework that is used to automate application testing with deep integration to z/OS and other test tools. The framework includes the following components:

1. Eclipse plug-in for developing tests.
2. Test managers: Standard programs that simplify the setup of environments that the application needs for testing, which is especially helpful for interacting with z/OS Batch and other middleware.
3. Test Runner: Performs the running of tests and drives the test lifecycle.

Galasa Java based programs that are called Test Managers can be used to programmatically interface to systems and middleware for both z/OS and distributed use cases. Galasa is especially helpful for interacting with z/OS systems and middleware because programs like the z/OS Batch Manager can submit, monitor, and retrieve jobs. Developers and testers can use these managers, augment them, and develop new managers. Galasa can be used to automate different integration test scenarios that involve z/OS applications, and it can be extended to combine tests that are created with tools like Rational Integration Tester or Selenium to bring different tests for an application, z/OS and distributed, to a single place.



Planning for an implementation

We introduced the types and levels of testing that organizations perform, the best practices to achieve agile and continuous testing, and the tools that can be used to accelerate testing.

The rest of this paper delves primarily into developer-led testing. This chapter describes the installation topology and the configuration steps for IBM z/OS Automated Unit Testing Framework (ZUnit) and IBM Z Virtual Test Platform (ZVTP). Both products share a host component that is called IBM Dynamic Test Runner (DTR), which forms the base of recording and stubbing for various middle-ware. The installation topology and setup are slightly different.

Figure 3-1 shows the installation view of ZUnit and ZVTP.

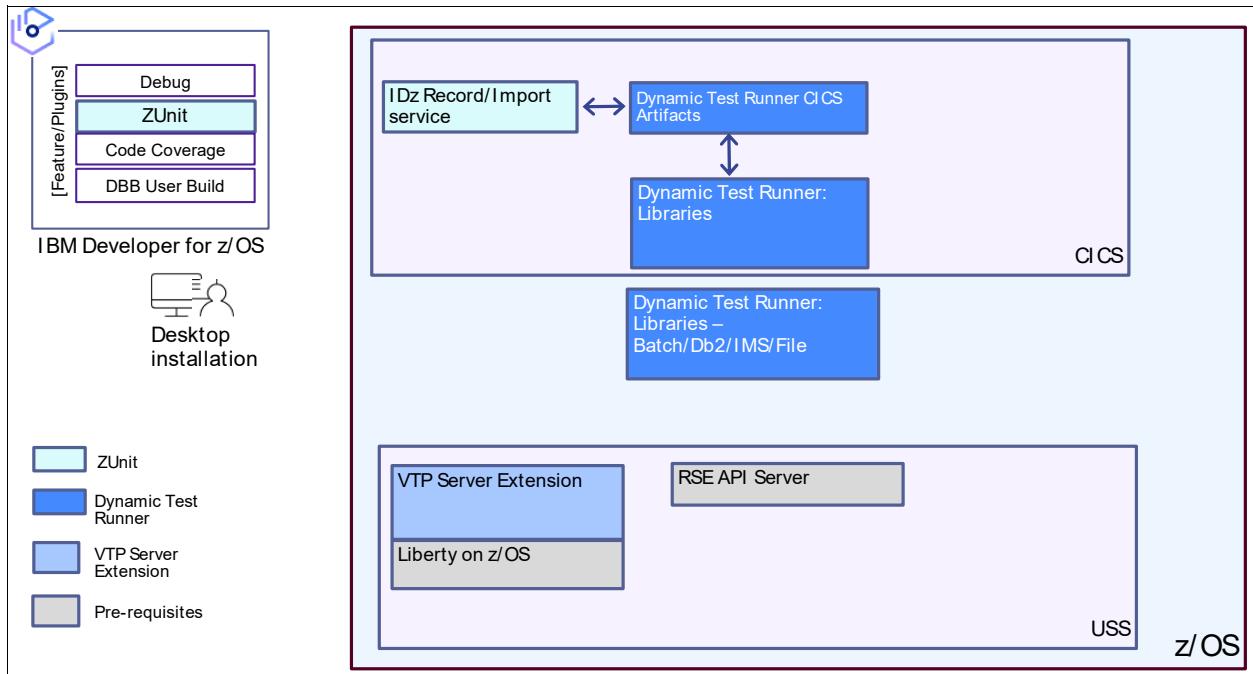


Figure 3-1 Installation view for ZUnit and ZVTP

3.1 Installing ZUnit

The following components must be installed to use ZUnit by using the IBM Developer for z/OS (IDz) Client. The client can be installed by using an Installation Manager or as a P2 Eclipse installation. At the time of writing, ZUnit is supported on Windows machines.

IDz consists of many function modification identifiers (FMIDs), but the following ones are mandatory for ZUnit:

- ▶ z/OS Explorer
- ▶ z/OS Explorer Extensions
- ▶ DTR
- ▶ z/OS Debugger
- ▶ z/OS Source code analysis
- ▶ Dependency Based Build (DBB) (for source code in Git)

3.2 Configuring ZUnit

There are two parts that must be configured for ZUnit:

- ▶ The client side, which runs in IDz. The client is configured by a Build or Tools administrator.
- ▶ The host side, which is configured by a system administrator in collaboration with an IBM CICS, Db2, or IMS administrator.

3.2.1 Configuring the host-side components

The easiest and recommended way to configure ZUnit is by going to [Host Configuration Assistant for Z Development \(HCA\)](#). This tool is a no-charge tool that lists the steps to perform as part of configuring any product. ZUnit is part of the IDz product, so it can be used with any of the portfolio of products that are built on IDz, such as IBM Developer for z/OS Enterprise Edition (IDzEE), IBM Application Delivery Foundation for z/OS (ADFz), or even IDz.

An example of using HCA to configure ZUnit if ZUnit is installed with ADFz is shown in the following steps. The host configuration for ZUnit consists of configuring components from the z/OS Host Explorer extensions and the DTR. Although the steps are not covered by these screen captures, it is a best practice that the appropriate configuration steps are followed for Debug and z/OS Source Code Analysis to debug a test case and run code coverage. You also should perform the configuration steps for IBM DBB because Git is recommended as the Source Control Management (SCM) to store unit tests.

1. Select the products to install (see Figure 3-2).

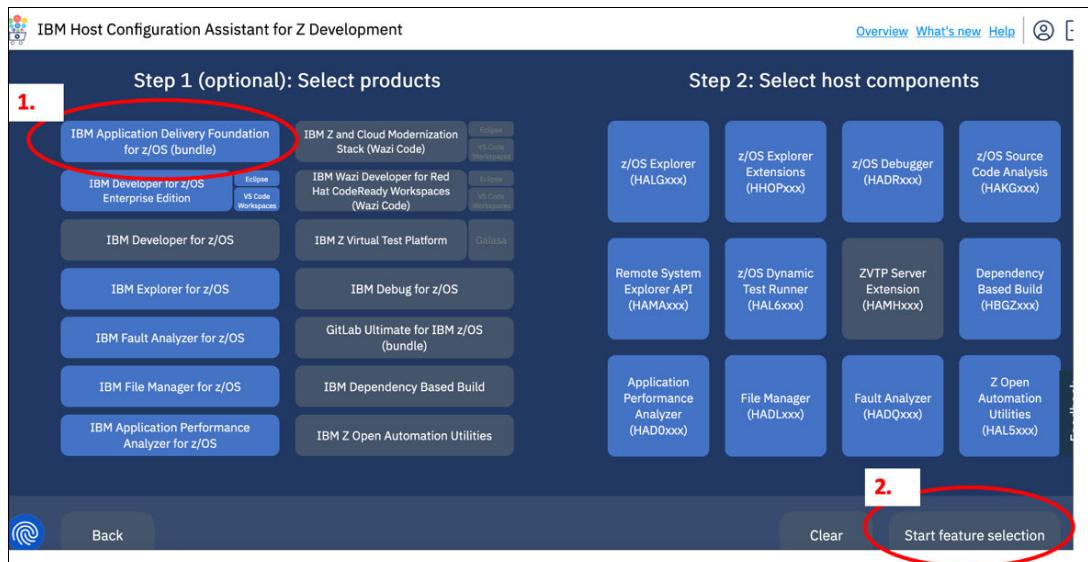


Figure 3-2 Selecting the product to install

2. Select XUnit support from z/OS Explorer Extensions (Figure 3-3).

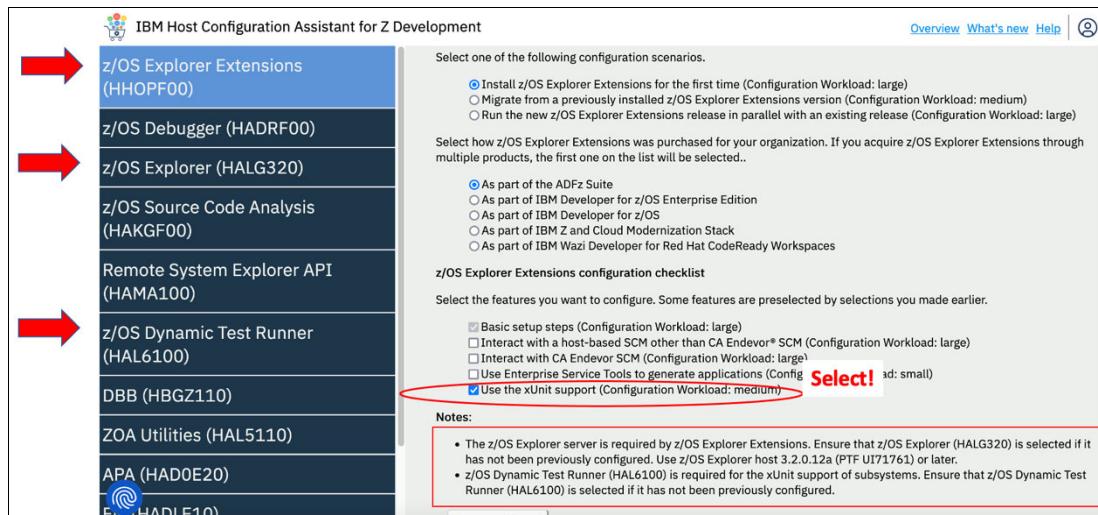


Figure 3-3 Selecting FMIDs

3. Select the setup steps from z/OS DTR (Figure 3-4).

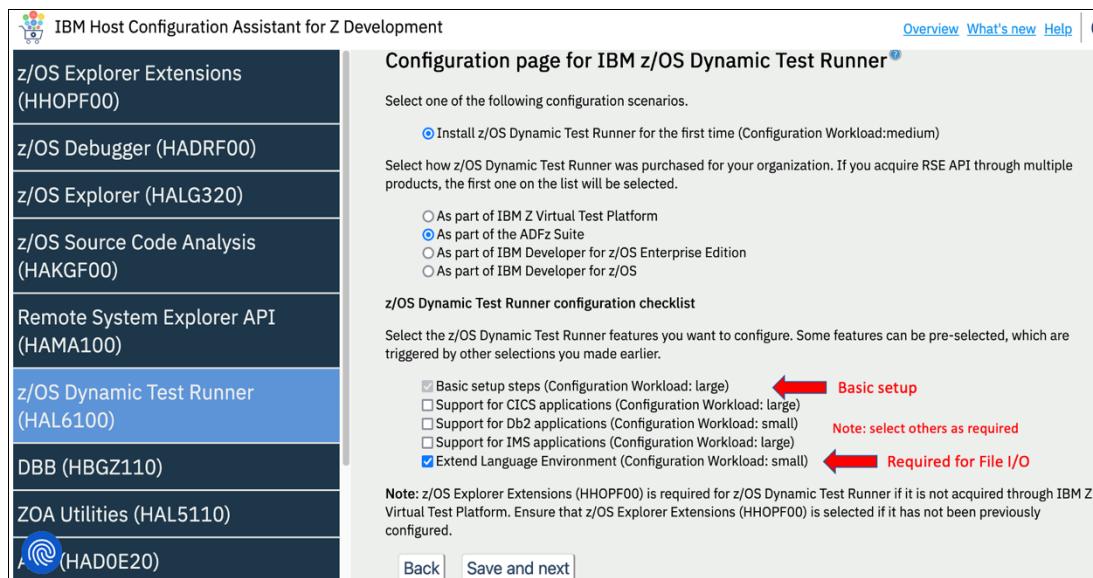


Figure 3-4 Selecting the DTR features

4. Generate the report.

HCA generates a summary of the steps that are required to configure the different components after completing all selections within the tool. The summary of the configuration steps on the host side for ZUnit are shown in Table 3-1.

Table 3-1 IBM Host Configuration Assistant: generated report of configuration steps summary

Step	FMID or host component	Task
1	z/OS Host Explorer Extensions	CICS system initialization parameter (SIP) updates: ► TCPIP=YES ► USSHOME=/usr/lpp/cicsts/cicsts56
2	z/OS Host Explorer Extensions	CICS JCL updates ► REGION=0M ► Add SFELLOAD to DD DFHRPL.
3	z/OS Host Explorer Extensions	CICS CSD updates ► Customize and submit SFELSAMP (AZUCSD). ► Customization includes changes to the AZUCREST definition. AZUCREST, which is a part of the ZUnit recording control web services, must be defined as remote in your web owning regions and point at a region to be used as a central point of control, where AZUCREST should be defined as local.
4	z/OS Host Explorer Extensions	Security updates: Allow users to create, update, and delete TDQUEUE that starts with ZU*. The CICS region and the users should have access to the data set qualifiers that are used to create the queue.
5	DTR	Submit the BZUSSETUP setup job.
6	DTR	Update PARMLIBs: ► Define SBZULINK as APF-authorized in PROGxx. ► Define SBZULINK to LINKLIST definitions in PROGxx.

Step	FMID or host component	Task
7	DTR	Add members (BZUINCL and BZUP*) to PROCLIB.
8	DTR	Add security definitions with BZURACF.
9	DTR	Update CICS SIPs. RENTPGM=NOPROTECT captures data on nonstandard CICS interfaces.
10	DTR	Update CICS region JCL: ► REGION=0M ► Add SBZULOAD to DD DFHRPL.
11	DTR	Define resources to CICS. Use a sample JCL BZUCSD with the following changes: 1. The BZUQ definition can be omitted. 2. BZUC should be defined as local in all regions. 3. BZUVFILE can either be: a. Local in one region and remote in all others. b. Defined as local in all regions if BZUVSAM is updated to define the data set as RLS.
12	DTR	Relink Db2 load modules in SBZURES (sample JCL BZULDB2). Make sure that this data set is separate from SBZULOAD.
13	DTR	Add SBZUSAMP(BZUPDB2) to PROCLIB (This task is already done if the wildcard in step 7 in this table included everything.)
14	DTR	APF-authorize SBZURES if it is added to a STEPLIB that holds Db2 load libraries.
15	DTR	Ensure that SBZURES is ahead of the Db2 SDSNLOAD wherever Db2 recording is required, for example, STEPLIB of IMS control region.
16	DTR	Extend the IBM Language Environment® for recording and replaying file I/O by using the sample JCL BZULLE.
17	DTR	Configure the BZUCFG data set (the data set that is referred to by TDQ BZUC) that is referenced by the controlling region (as selected in step 3 in this table) to specify which regions to start recording in when a user request is received. The new tags are: <pre><runner:connection group="csysgrp" cicsplex="cicsplex" /> <runner:recording target="applids"/></pre> ► "csysgrp" is the name of a CICS System Group that contains the required regions. ► "cicsplex" is the name of the IBM CICSplex® that contains "csysgrp". ► "applids" is a comma-separated list of IBM VTAM® applids with optional wild cards ("?" for a single character, and "*" for any number of characters).

Note: These steps do not include DBB, debug, or code coverage and might change as features are added to the product. The details are provided only for reference. Access the Host Configuration Assistant for the latest updates.

3.2.2 Configuring the client

The IDz client must be configured for unit testing with ZUnit. This section describes the configuration that is required to run ZUnit when it is part of a local Git or Engineering Workflow Management (EWM) project. These settings can be accessed opening Eclipse and selecting **Windows** → **Preferences** → **ZUnit**. You can set up the configuration in a workspace. The activities that are mentioned in this section are performed by a tools administrator and replicated across workspaces.

General settings

Confirm the runner as BZUPPLAY on the Preferences screen, as shown in Figure 3-5, which is for IDz 15.0.5.

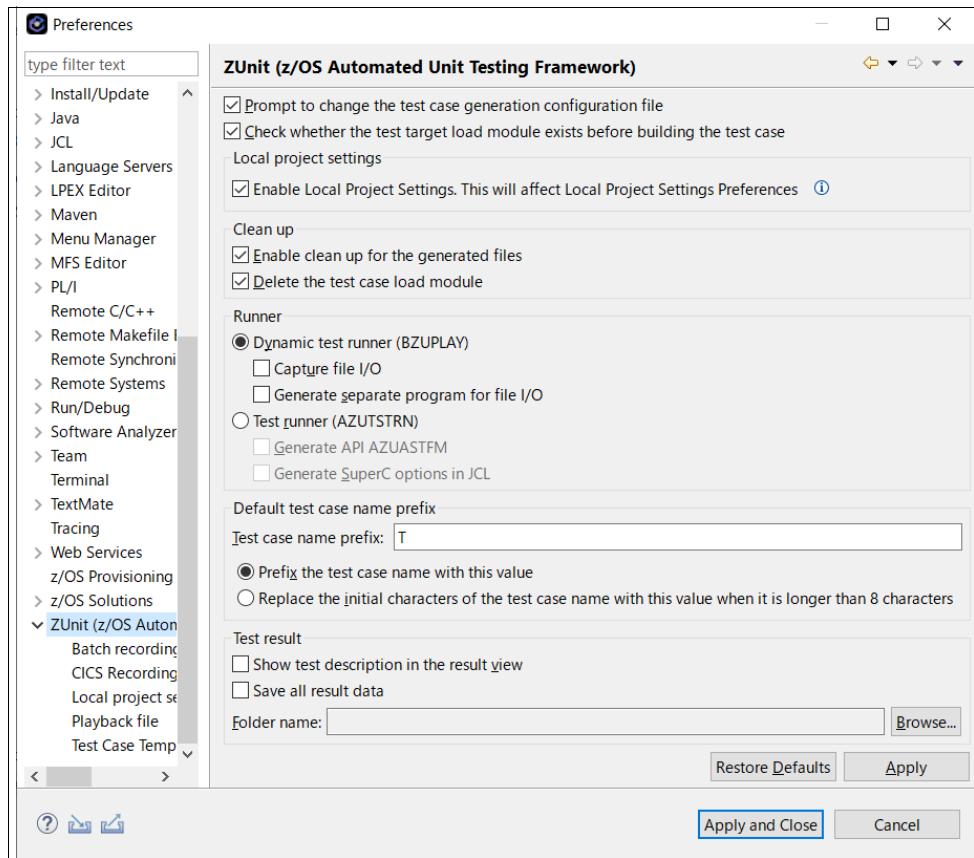


Figure 3-5 ZUnit preferences for IDz 15.0.5

If you are using if IDz 16.0.0, the preferences are simplified, as shown in Figure 3-6.

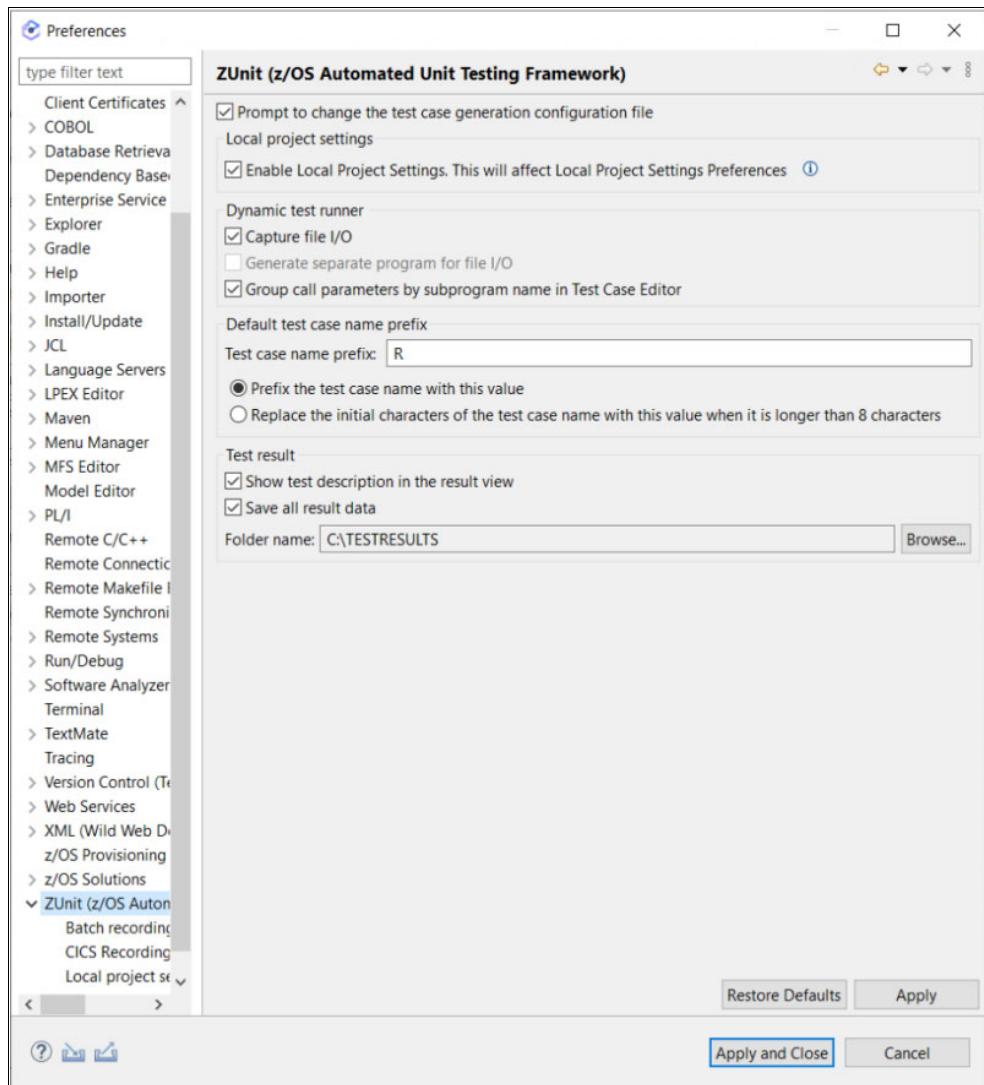


Figure 3-6 ZUnit preferences for IDz 16.0.0

Verify the default name for the playback file. The developer must have access to create and modify data sets with this High-Level Qualifier (HLQ). If you are working with CICS, the user and the CICS region must have access to create a Transient Data Queue with a data set name that starts with the HLQ that is specified in Figure 3-7.

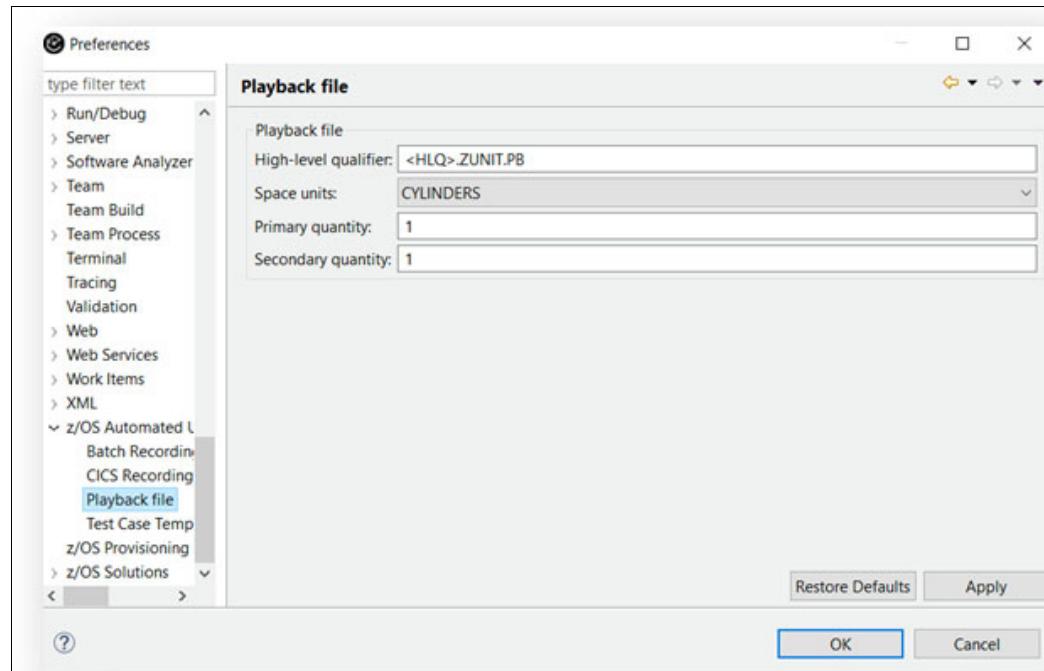


Figure 3-7 Preferences for the playback file

You can run a ZUnit test case with properties that are stored in the local project settings of a local project (Git project) or by using a remote property group. For running a ZUnit test case, concatenate the necessary load libraries containing the application under test into the STEPLIB within the remote property group or within the local project settings (see Figure 3-8 on page 25). Provide the proclib data set where the customized BZUPPLAY exists is in the **JCLLIB** statement in the JCL job card within the JCL tab of the local project settings or remote property group.

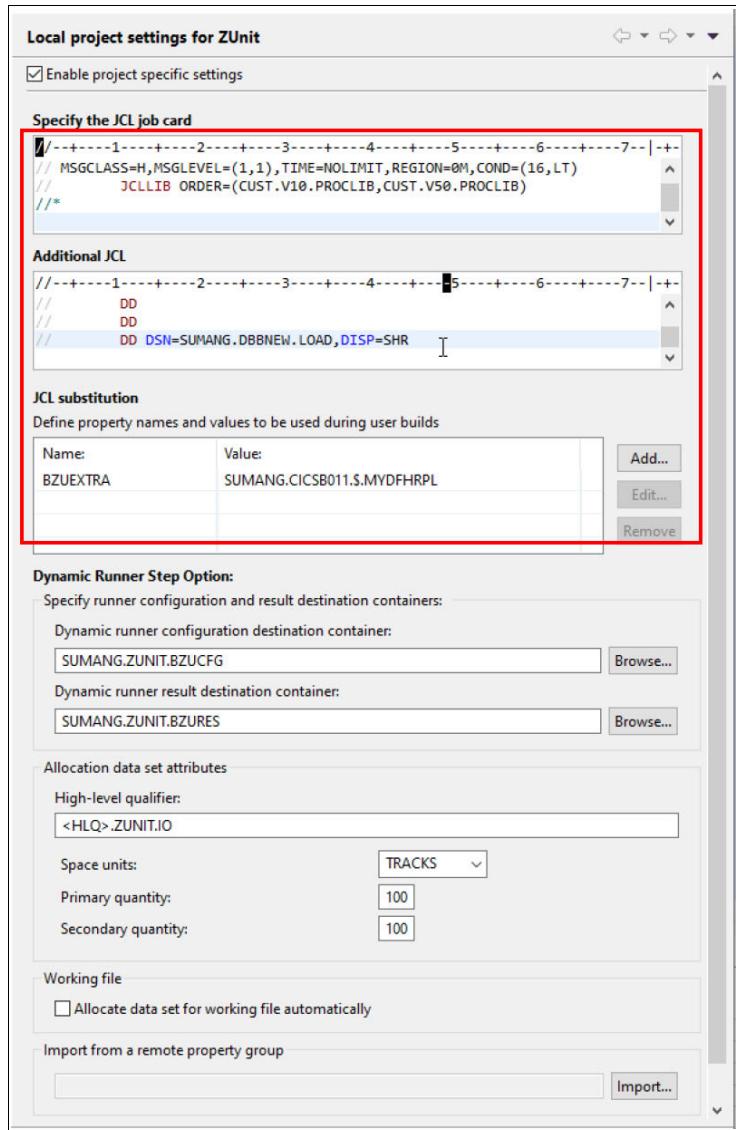


Figure 3-8 Local project settings

Batch settings

To create a batch test, configure the “Batch recording service” preferences. To set the dynamic runtime libraries for IDz 15.0.x and earlier, use <HLQ>.SBZULOAD and <HLQ>.SBZURES. If you use file I/O recording, add the relinked <HLQ>.SBZULLEP data set.

Figure 3-9 shows example settings for batch for IDz 15.0.x and earlier.

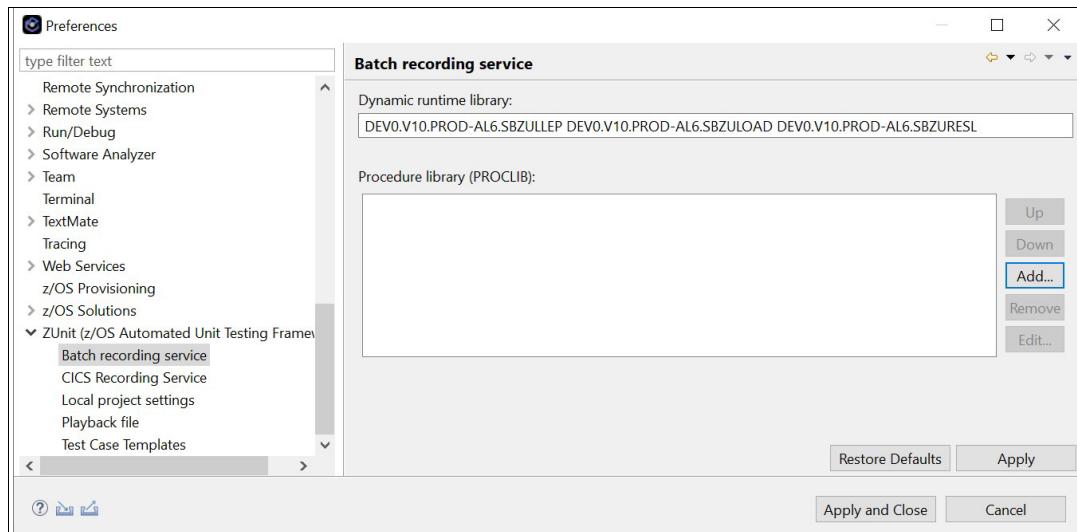


Figure 3-9 Example settings for batch for IDz 15.0.x

For IDz 16.0, the DTR runtime libraries do not need to be specified in Preferences, as shown in Figure 3-10. These libraries can be retrieved from the environment variable settings on the connected z/OS host where the testing occurs. Ensure that the system programmer has set the BZU variable in the ELAXF include member.

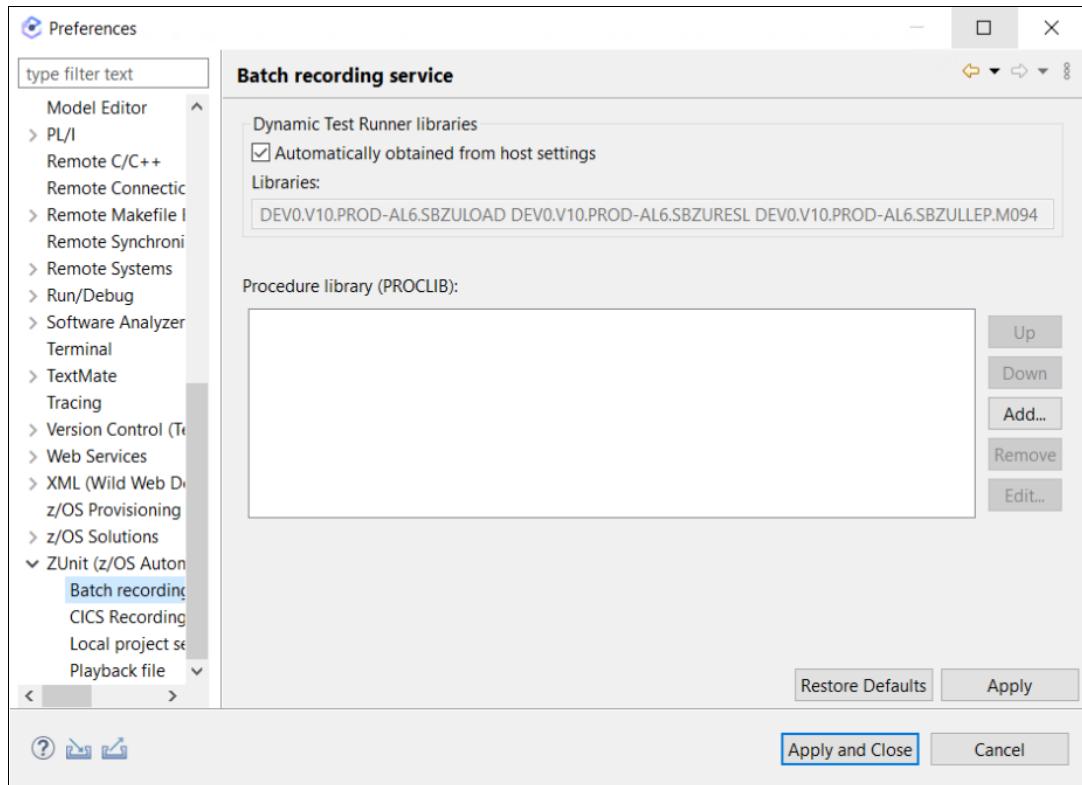


Figure 3-10 Example settings for batch for IDz 16.0.0

If you use input/output data sets, enable recording of the files by selecting the **Dynamic test runner (BZUPLAY)** checkbox, as shown in Figure 3-11. By selecting this checkbox, a unit test can be independent of files. (The SYSIN data set is not recorded).

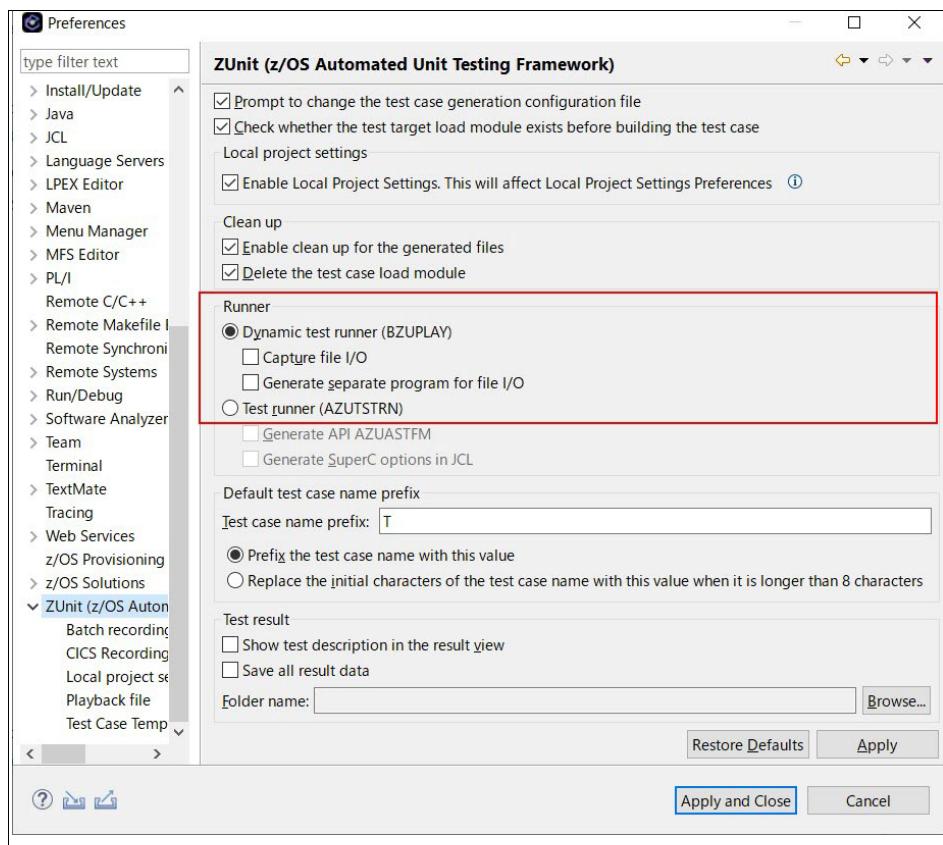


Figure 3-11 Selecting the DTR (BZUPLAY)

CICS settings

Set up the CICS recording service URL, as shown in Figure 3-12. This information is retrieved from the CICS administrator who configures the host side.

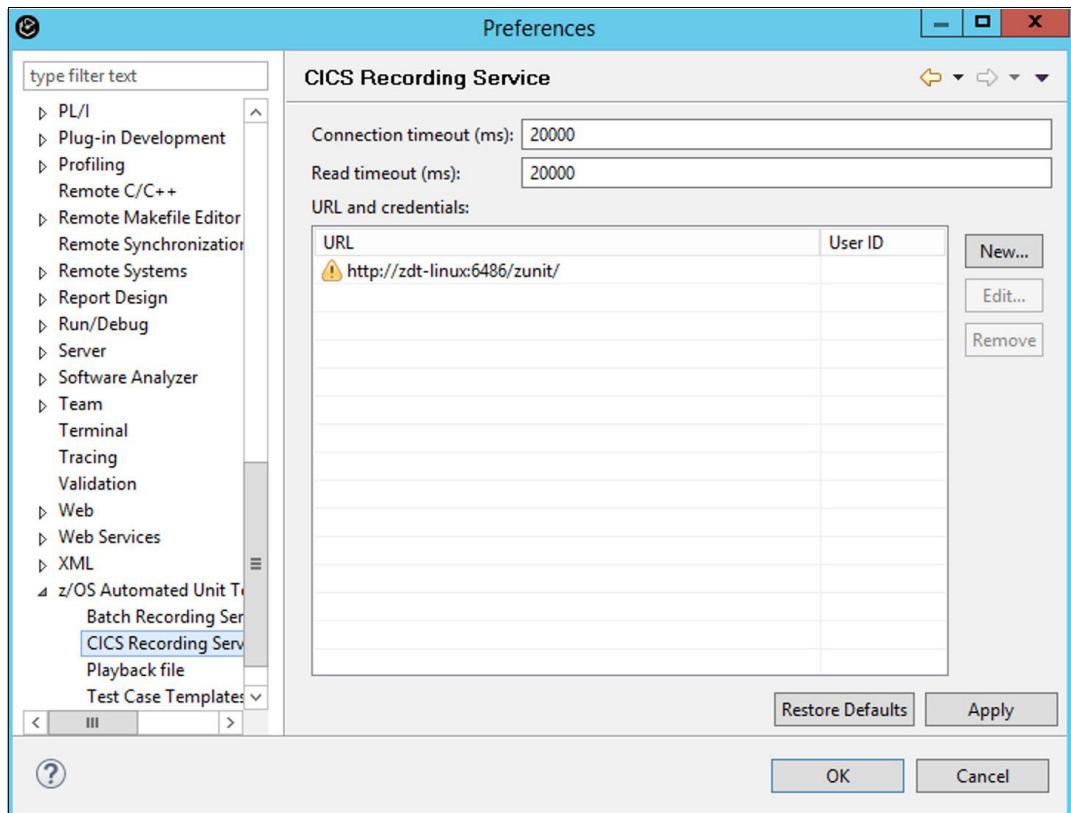


Figure 3-12 Preferences for CICS recording

Other prerequisites

For program-to-program calls to be captured by the recording feature and interpreted during the testing, the programs being tested should be compiled with RENT linked with the option REUS=RENT, which also implies the setting of the compilation option RENT (NORENT will not work).

To stub calls to subprograms, dynamically link them with the program that is being tested.

3.3 Installing IBM Z Virtual Test Platform

The main components that are required to install ZVTP are as follows:

- ▶ IBM DTR
- ▶ ZVTP Server Extension
- ▶ Prerequisites
- ▶ Liberty for z/OS
- ▶ Remote System Explorer API

3.4 Configuring the IBM Z Virtual Test Platform

As with ZUnit, the Host Configuration Assistant for IBM Z Development can be used to get the detailed list of steps that are required to configure ZVTP. Figure 3-13 shows the IBM Host Configuration Assistant for IBM Z Development home window. The red number 1 indicates that we selected the ZVTP as the product that we want to configure.

Note: This example does not include the configuration of the software that is installed as prerequisites.

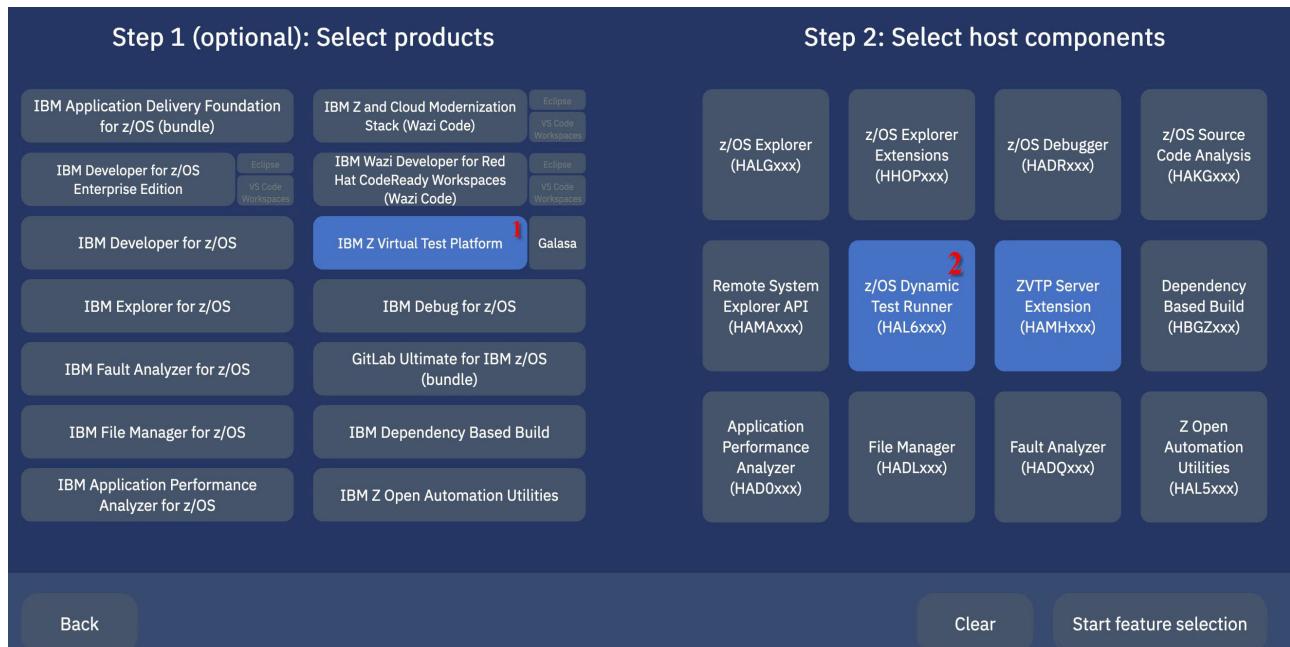


Figure 3-13 IBM Host Configuration Assistant for IBM Z development: Selecting the product to install

The following steps demonstrate the usage of the IBM Host Configuration Assistant for IBM Z Development to get the detailed list of steps that are required to configure ZVTP:

1. From the Host Configuration Assistant, select the tile that is labeled with the red number 2 in Figure 3-13 on page 29 to get the setup steps from the z/OS DTR based on the technologies that are involved (Figure 3-14).

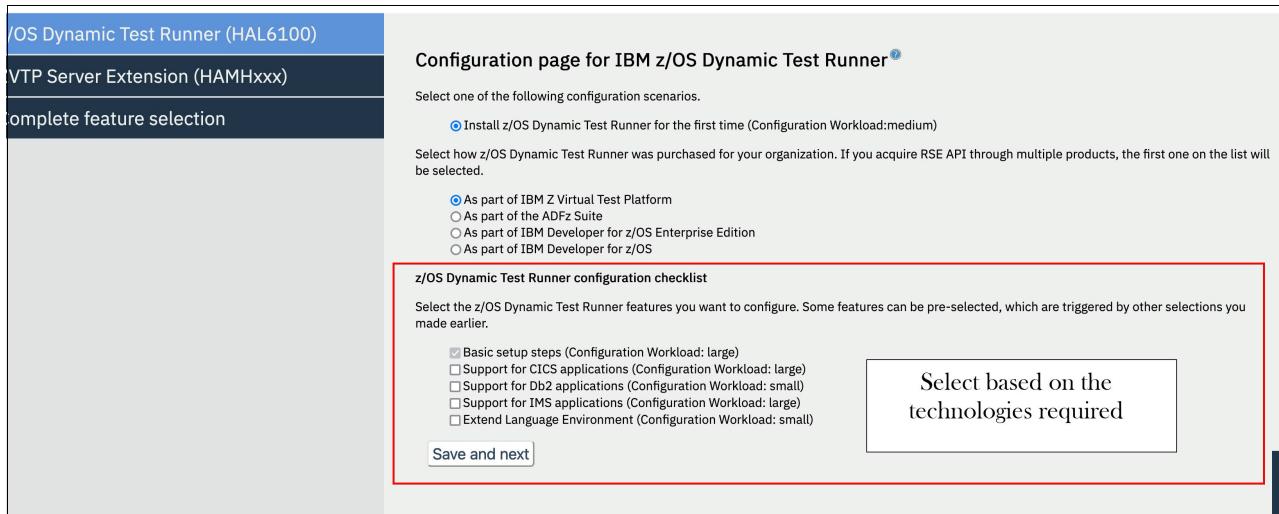


Figure 3-14 Selecting the z/OS DTR features that are needed for testing

2. Select the settings for the ZVTP Server Extension (Figure 3-15).

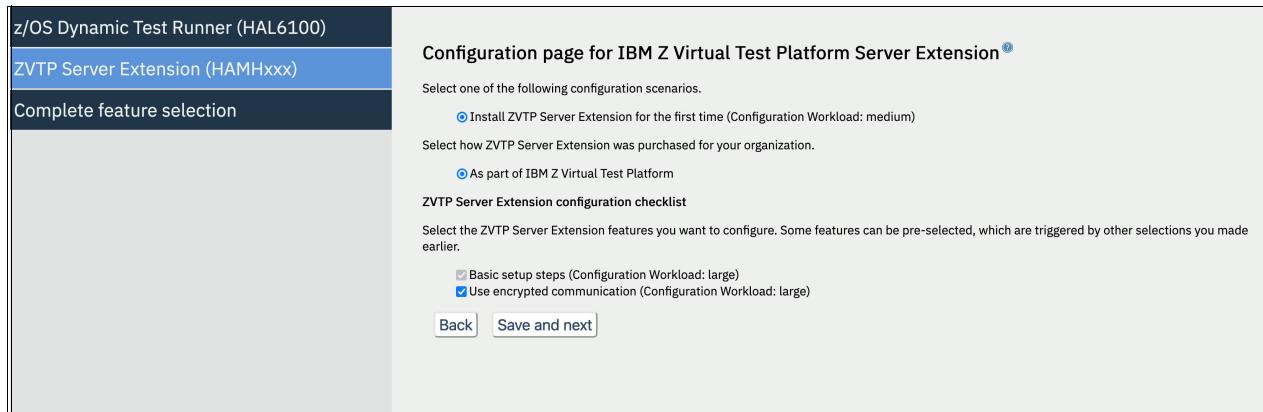


Figure 3-15 Selecting the ZVTP Server Extension

This action generates a checklist of items to be configured.

Table 3-2 lists a set of sample configuration steps that are generated. These steps are like ZUnit, but can contain additional technologies that are supported by ZVTP like IMS TM.

Table 3-2 Host Configuration Assistant generated report: Sample configuration steps that are generated

Step	FMID or host component	Task
5	DTR	Submit the BZUSETUP setup job.
6	DTR	<ul style="list-style-type: none"> ▶ Update PARMLIB. ▶ Define SBZULINK as APF-authorized in PROGxx. ▶ Define SBZULINK to LINKLIST definitions in PROGxx.
7	DTR	Add members (BZUINCL and BZUP*) to PROCLIB.
8	DTR	Add security definitions with BZURACF.
9	DTR	Update CICS SIPs. RENTPGM=NOPROTECT captures data on nonstandard CICS interfaces.
10	DTR	<ul style="list-style-type: none"> ▶ Update CICS region JCL. ▶ REGION=0M ▶ Add SBZULOAD to DD DFHRPL.
11	DTR	<ul style="list-style-type: none"> ▶ Define resources to CICS. ▶ Use Sample JCL BZUCSD with the following changes: <ul style="list-style-type: none"> – BZUQ definition can be omitted. – BZUC should be defined as local in all regions. – BZUVFILE can either be: <ul style="list-style-type: none"> • Local in one region and remote in all others. • Defined as local in all regions if BZUVSAM is updated to define the data set as RLS.
12	DTR	Relink Db2 load modules in SBZURES (Sample JCL BZULDB2). Make sure that this data set is separate from SBZULOAD.
13	DTR	Add SBZUSAMP(BZUPDB2) to PROCLIB. (This task is already done if the wildcard in step 7 in this table included everything.)
14	DTR	APF-authorize SBZURES if it is added to a STEPLIB that holds Db2 load libraries.
15	DTR	Ensure that SBZURES is ahead of Db2 SDSNLOAD wherever Db2 recording is required, for example, STEPLIB of the IMS control region.
16	DTR	Extend the Language Environment for recording and replaying file I/O by using the sample JCL BZULLE.

Step	FMID or host component	Task
17	DTR	<p>Configure the BZUCFG data set (the data set that is referred by TDQ_BZUC) that is referenced by the controlling region (as selected in step 3 in Table 3-1 on page 20) to specify which regions to start recording in when a user request is received.</p> <p>The new tags are as follows:</p> <ul style="list-style-type: none"> ▶ <runner:connection group="csysgrp" cicsplex="cicsplex" /> ▶ <runner:recording target="applids"/> <p>(You can specify either one.)</p> <ul style="list-style-type: none"> – "csysgrp" is the name of a CICS System Group that contains the required regions. – "cicsplex" is the name of the CICSplex that contains "csysgrp". – "applids" is a comma-separated list of IBM VTAM applids with optional wild cards ("?" for a single character, and "*" for any number of characters).
18	ZVTP Server Extension	Submit the BZ0SETUP setup job.
19	ZVTP Server Extension	Reserve TCP/IP port 6700.
20	ZVTP Server Extension	Submit the BZ0RACF security setup job.
21	ZVTP Server Extension	Add members (BZ0INCL, BZ0PJVM, and BZ0SRVR) to PROCLIB.
22	ZVTP Server Extension	<p>Update PARMLIB:</p> <ul style="list-style-type: none"> ▶ Mount the file system in BPXPRMxx. ▶ Start the BZ0SRVR started task in COMMNDxx.
23	ZVTP Server Extension	Customize /etc/bzo/bzo.env.
24	ZVTP Server Extension	Customize /etc/bzo/vtpRestServer.cfg.
25	ZVTP Server Extension	Customize /etc/bzo/dbmsIsDerby.
26	ZVTP Server Extension	Run installation verification.
27	ZVTP Server Extension	<p>Set up AT-TLS:</p> <ol style="list-style-type: none"> 1. Set up the syslogd service. 2. Configure AT-TLS in PROFILE.TCPIP. 3. Create PAGENT, which is the Policy Agent started task. 4. Create /etc/pagent.conf, which is the Policy Agent configuration file. 5. Perform related security updates.
28	ZVTP Server Extension	Create a key ring and a server certificate for client communication.
29	ZVTP Server Extension	Create a key ring and load the IBM Remote System Explorer (RSE) API certificate for RSE API communication.
30	ZVTP Server Extension	Add a Tunneled Transport Layer Security (TTLS) policy for client communication.
31	ZVTP Server Extension	Add a TTLS policy for RSE API communication.
32	ZVTP Server Extension	Activate TTLS policies.

Note: These steps might change as features are added to the product. The details are provided only for reference. For more information, see the Host Configuration Assistant.



A sample scenario

This section takes a sample change scenario and explains how z/OS Automated Unit Testing Framework (ZUnit) and IBM Z Virtual Test Platform (ZVTP) can help to test the application change. This example uses a General Insurance Application (GenApp), which is an application in Insurance Company AXZ. GenApp enables business to create policies for motor insurance, endowment, home insurance, and commercial property. GenApp also allows business operations to inquire about customer profiles and add customers. GenApp has a Tn3720 interface for interfacing with the application by using a basic mapping support (BMS) map. GenApp also has certain data like customer information and policy details that can be exposed through APIs. These APIs are consumed by web pages, chatbots, and various other digital media, primarily to inquire about and renew insurance policies.

The current customer policy API does not display the vehicle registration number, so customers that do registration renewals through chatbots have requested that the registration number is displayed. To this task, you must change the COBOL application the API that is exposed on the screen. This chapter describes the tools and the processes to shift left and test this change early in the development cycle.

Figure 4-1 shows an example of the flow of a transaction SSP1 through the application.

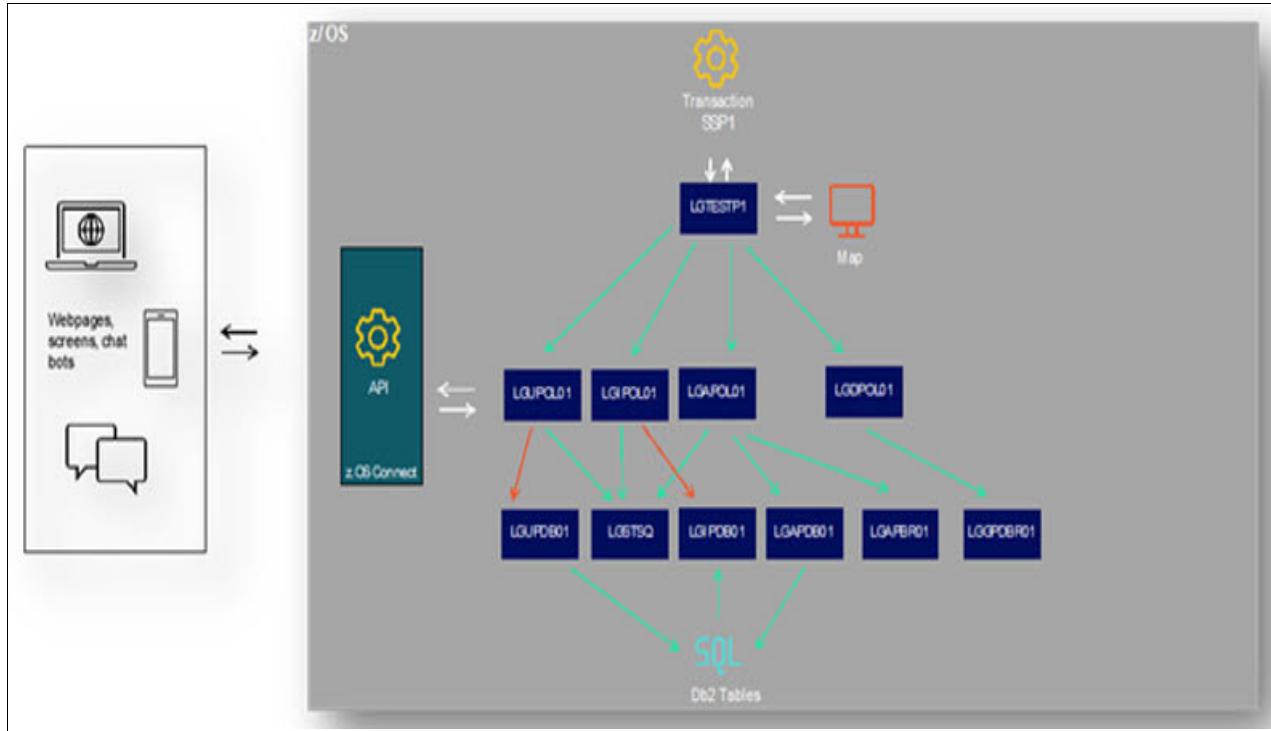


Figure 4-1 Architecture of the application under test

When the development team receives a request for a new GenApp feature or business enhancement, the first task is to identify the areas of code to change. This task normally involves debugging or using analysis tools like [IBM Application Discovery and Delivery Intelligence \(ADDI\)](#). After the programs that are involved in the change are identified by either a developer or an architect, the changes to the code can be made. Referring to Figure 4-1, assume that the developer Deb is assigned an issue that details the program flow SSP1 -> LGIPOL01 -> LGIPDB01, and the primary program to change, which is LGIPDB01.

4.1 Testing and understanding the transaction flow of the application with the IBM Z Virtual Test Platform

The ZVTP platform helps to create a test case to test a transaction or a batch program flow. This test can be replayed without the subsystem to understand the impact of a change. ZVTP can create a test that involves the COBOL or PL/I components in the application. This test can be created by recording data from the API invocation or from BMS map, which creates a test for the GenApp SSP1 transaction that is shown in Figure 4-1.

Figure 4-2 on page 37 shows how to create a ZVTP test by enabling a recording of a test that uses the IBM CICS transaction BZUS.

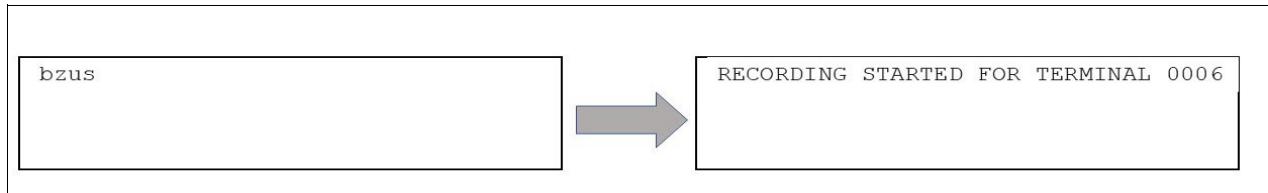


Figure 4-2 Starting a recording for a CICS transaction

After the record is enabled, the transaction that is tested is invoked from a 3270 emulator, by using a chatbot, or by using an API tool (Figure 4-3).

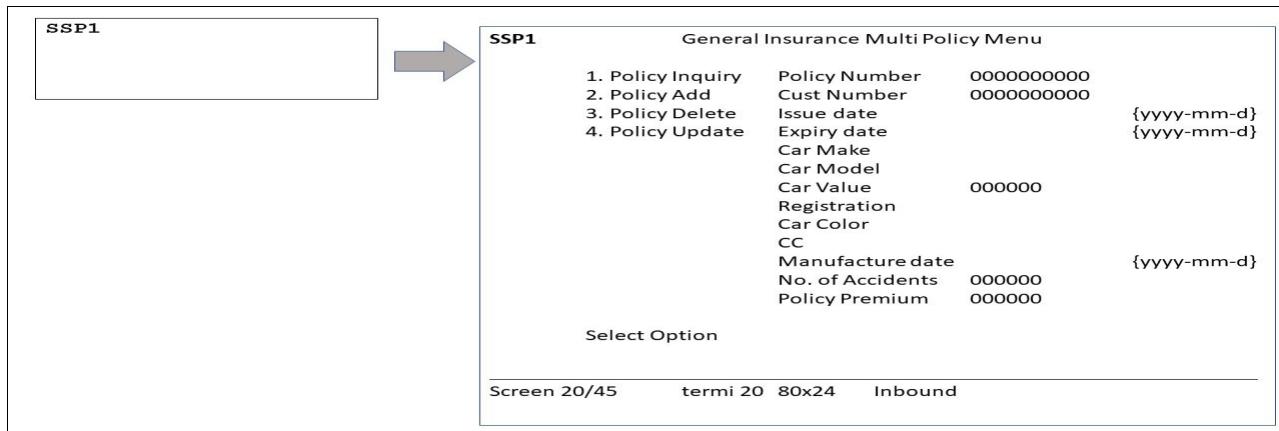


Figure 4-3 Issuing the transaction

A window opens and shows the application information, as shown in Figure 4-4.

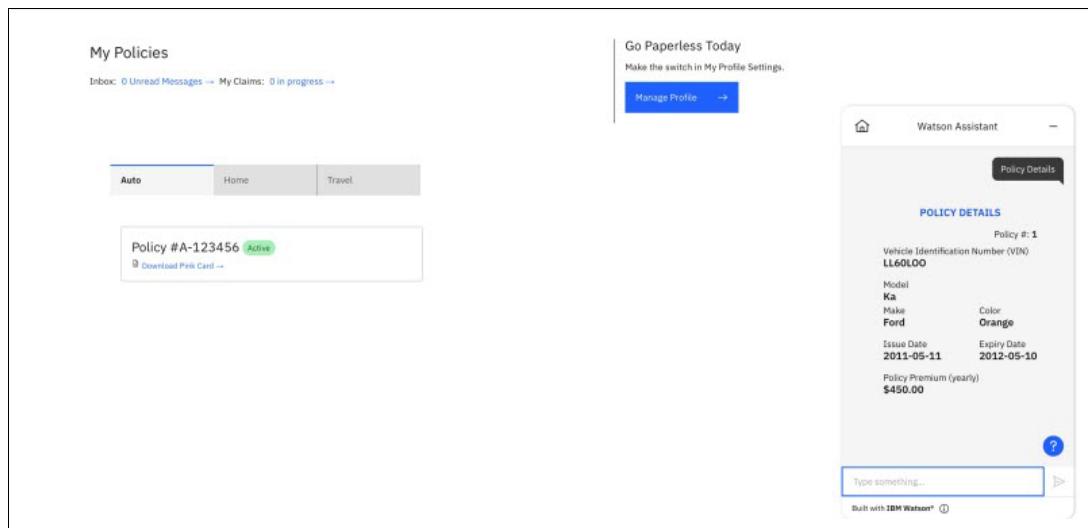


Figure 4-4 Application information

After running the transaction or the flow, the recording is stopped with a **BZUE** command, as shown in Figure 4-5. The Policy API shows the vehicle registration number instead of the 20-char vehicle engine.

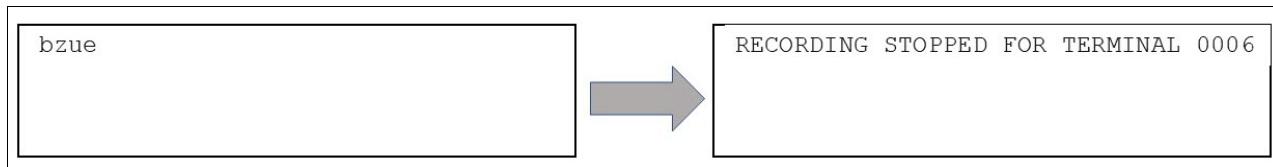


Figure 4-5 Terminating the recording

The recording is extracted and placed into any data set by using a **BZUW** command.

The recorded test can be imported and viewed in the ZVTP web UI (Figure 4-6), which helps the developer understand the flow of the transaction and obtain a baseline test if there is no ZVTP test for this transaction or functional flow. The test case details the programs and components that are involved in the test.

IBM Z Virtual Test Platform

Test cases

Motor Policy Green screen

Test case details

Name	Motor Policy Green screen
Created by	IBM Corporation
Description	Enter test case description

Recording data set: SUMANG.ZUNIT.PB.LGIPDB01
Recording time: 3/19/2022, 2:50:00 PM
Type: CICS

Run test case Save changes View less

Table Content

Q Enter the string to filter the tables

- Transaction (SSPs)
 - Db2 Subsystem subsystems (1)
 - DSNO
 - Maps (1)
 - SSMAPP1
 - Mapsets (1)
 - SSMAP
 - Programs (3)
 - LGIPDB01
 - LGIPOL01
 - LGTESTP1

Figure 4-6 Table view of the SSP1 transaction

The timeline view (Figure 4-7 on page 39) in the test case shows the flow of the transaction and the statements that ran. Deb can understand the programs that are involved and the inner workings of the test. They also can verify the impact of their changes.

Figure 4-7 ZVTP timeline view of the SSP1 transaction

4.2 Testing single programs by using unit testing

To ensure that the right data is shown in the window, the VIN number must be retrieved from the database and the program LGIPDB01. Before the change is done, check the repository to see whether there is a unit test for the program. If there is not one, you must create a unit test.

4.2.1 Creating a ZUnit test

With ZUnit, a developer can test a single program without it being impacted by any external variables. A unit test can be created for a program by invoking the z/OS Automated Unit Testing Framework feature from IBM Developer for z/OS (IDz), as shown in Figure 4-8.

Figure 4-8 Creating a ZUnit test case

With the unit test, Deb can verify the values of the data structures at I/O points within the program. ZUnit parses the program and extracts the data structures that are used within the I/O statements in the program, such as Linkage, Commarea, EXEC CICS, and EXEC SQL. Instead of creating the input and expected values at each I/O statement, Deb can easily create a sample test by recording from an actual run, as shown in Figure 4-9.

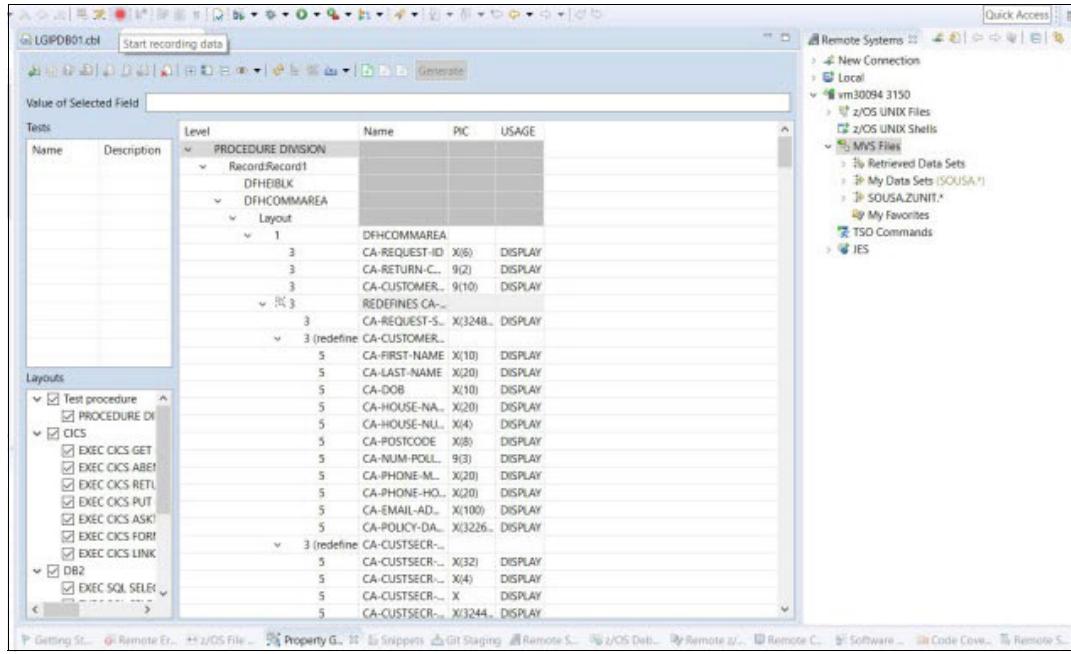


Figure 4-9 Recording the test

The recorded data is imported into a test case that is mapped to the input and the expected output (Figure 4-10). For example, the variables within the Commarea that are populated during the invocation of the program are tagged as input, and all the values that are passed back to the Commarea from the program are tagged as output. Similarly, in an SQL SELECT statement, the variables that are selected are an input to the program, and any variables in the WHERE clause are marked as output.

The screenshot shows the Rational Test Studio interface with the following details:

- Title Bar:** "LGIPDB01.cbl" and "LGIPDB01" are open.
- Toolbar:** Standard file, edit, and navigation icons are visible.
- Value of Selected Field:** A search bar labeled "Value of Selected Field".
- Tests:** A table showing a single entry named "TEST2" with a description of "TEST2".
- Level:** A hierarchical tree view of the program structure:
 - PROCEDURE DIVISION
 - Record:Record1
 - DFHIEBLK
 - DFHCOMMAREA
 - Layout
 - 1
 - 3
 - 3
 - 3
 - REDEFINES CA-REQ...
 - 3 (redefines)
 - 3 (redefines)
 - 3 (redefines)
 - CA-REQUEST-SPEC... X(32482)
 - CA-CUSTOMER-RE...
 - CA-CUSTSECR-REQ...
 - CA-POLICY-QUEUE...
 - CA-POLICY-NUM 9(10)
 - CA-POLICY-COM...
 - CA-ISSUE-DATE X(10)
 - CA-EXPIRY-DATE X(10)
 - CA-LASTCHANGED X(26)
 - CA-BROKERID 9(10)
 - CA-BROKERSREF X(10)
 - CA-PAYMENT 9(6)
- Layouts:** A list of recorded layouts:
 - Test procedure
 - PROCEDURE DIVISION
 - CICS
 - EXEC CICS GET CONTAINER
 - EXEC CICS ABEND
 - EXEC CICS RETURN
 - EXEC CICS ASKTIME
 - EXEC CICS FORMATTIME
 - EXEC CICS LINK [LGTSQ]
 - DB2
 - EXEC SQL SELECT INTO [POLICY,ENDOWME
 - EXEC SQL SELECT INTO [POLICY,HOUSE]
 - EXEC SQL SELECT INTO [POLICY,MOTOR]
 - EXEC SQL SELECT INTO [POLICY,COMMERC
 - EXEC SQL SELECT INTO [POLICY,COMMERC
 - EXEC SQL OPEN [Cust_Cursor,Zip_Cursor,C
 - EXEC SQL CLOSE [Cust_Cursor,Zip_Cursor,C
 - EXEC SQL FETCH [Cust_Cursor,Zip_Cursor]
 - EXEC SQL SELECT INTO [POLICY,CLAIM]

Figure 4-10 Importing the data after the test

After the data is populated, the test can be duplicated, and the values can be changed. ZUnit can generate the test case, which is a COBOL program that can be maintained within a Source Control Management (SCM) tool like Git. The various artifacts that are linked to the test case can be checked into the SCM (Figure 4-11).

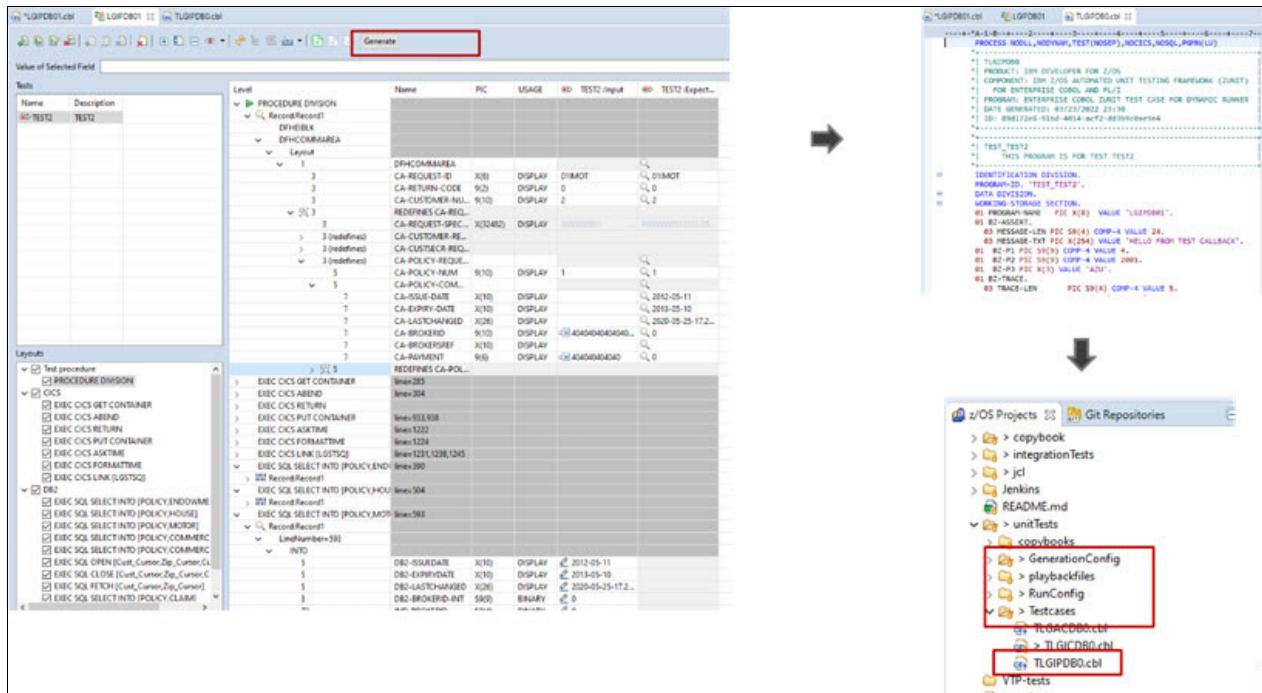


Figure 4-11 Test information in Git

The test case can be built like any other program, as shown in Figure 4-12 on page 43.

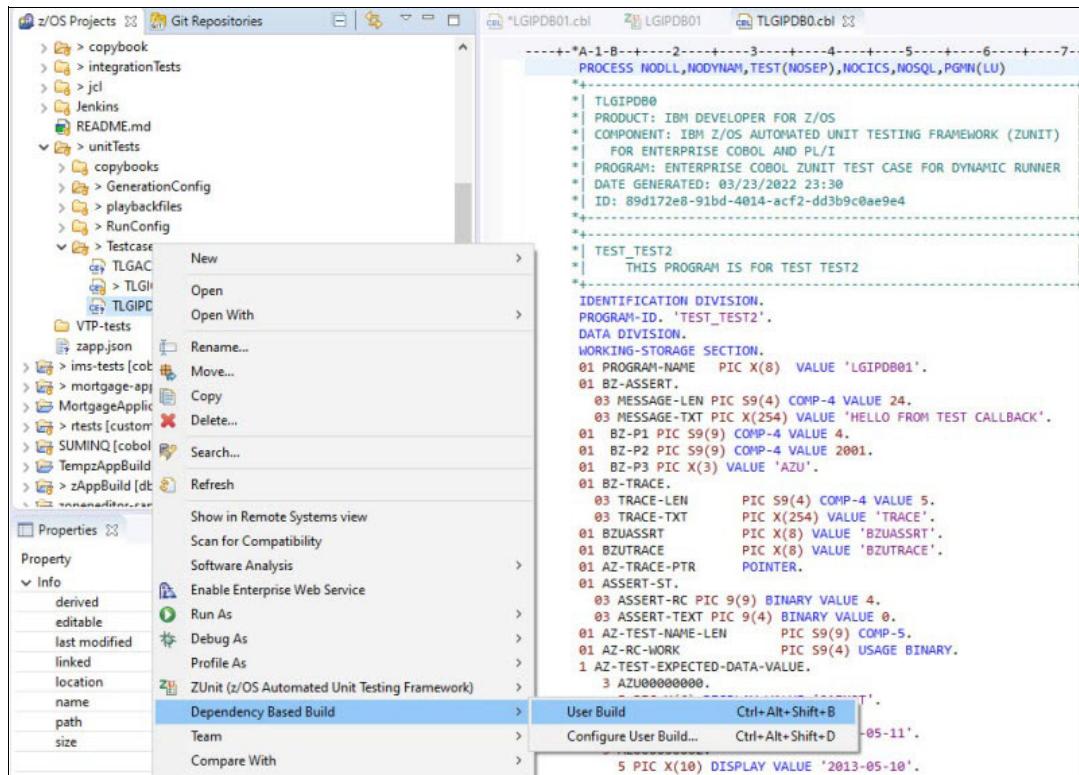


Figure 4-12 Test case build with Dependency Based Build

The test can be run with personal load libraries (Figure 4-13). The test invokes the test runner that runs the program without the middleware. The data for the calls to CICS or Db2 is taken from the recording or from the edited values within the test case.

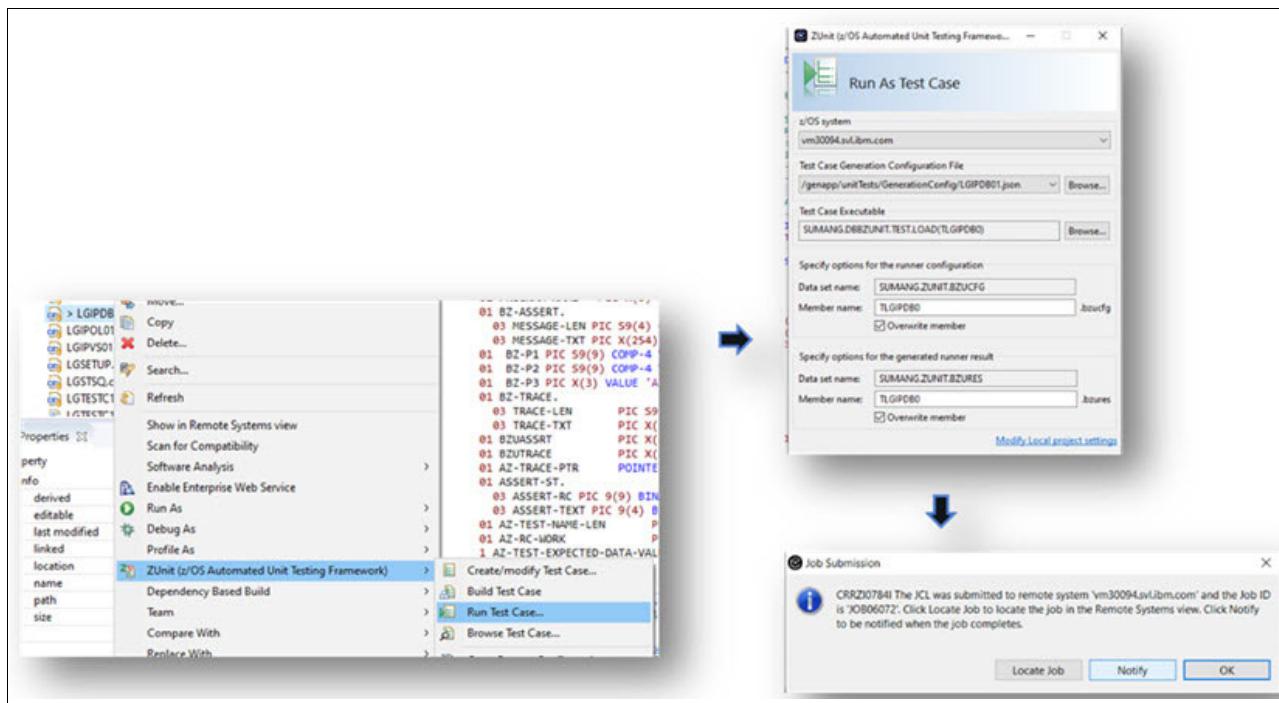


Figure 4-13 Running the test case

Once a unit test exists, Deb can proceed with their code changes.

4.2.2 Making application changes

A new field is added into the copybook structure that is used within the transaction to hold the VIN number. The SQL statement is edited to retrieve the VIN number from the database. Figure 4-14 shows the update to the application.

The figure displays two side-by-side code editors. The left editor shows an RPGLE program (LGPDB01.cbl) with various sections of code, some of which are highlighted with red boxes. The right editor shows a copybook (LGCMAREA.cpy) with sections for House policy, Motor policy, and Commercial policy, also with some code highlighted with red boxes.

Figure 4-14 Updating the application

4.2.3 Testing the program before commitment

Two tests are conducted before a program is committed:

- ▶ Unit tests
- ▶ Stub integration or impact tests

Unit tests

After the changes are made, Deb can test the program from their personal load libraries. They can edit the unit test (Figure 4-15 on page 45) to add the new field into the selected values, and update the unit test.

The screenshot shows the Rational Test Studio interface. On the left, there's a tree view under 'Tests' with a single entry 'TEST2'. Under 'Layouts', several checkboxes are checked, including 'Test procedure', 'PROCEDURE DIVISION', 'CICS' (with sub-options like 'EXEC CICS GET CONTAINER'), and 'DB2' (with sub-options like 'EXEC SQL SELECT INTO [POLICY,ENDOWMENT]'). The main right pane is a table titled 'Level' with columns: Name, PIC, USAGE, TEST2 :Input, and TEST2 :Expect...'. The table lists numerous test steps, many of which have their rows highlighted with a red border. One specific row, 'DB2-M-ACCIDENT', is highlighted with a red box.

Level	Name	PIC	USAGE	TEST2 :Input	TEST2 :Expect...
EXEC SQL SELECT INTO [POLICY,ENDOWMENT]					
> Record:Record1					
EXEC SQL SELECT INTO [POLICY,HOUSE]					
> Record:Record1					
EXEC SQL SELECT INTO [POLICY,MOTOR]					
> Record:Record1					
EXEC SQL SELECT INTO [POLICY,COMMERCIAL]					
> Record:Record1					
EXEC SQL OPEN [Cust_Cursor,Zip_Cursor,Customer]					
> Record:Record1					
EXEC SQL CLOSE [Cust_Cursor,Zip_Cursor,Customer]					
> Record:Record1					
EXEC SQL FETCH [Cust_Cursor,Zip_Cursor,Customer]					
> Record:Record1					
EXEC SQL SELECT INTO [POLICY,CLAIM]					
> Record:Record1					
EXEC SQL FETCH [CusClaim_Cursor]					
INTO					
5	DB2-ISSUEDATE	X(10)	DISPLAY	2012-05-11	
5	DB2-EXPIRYDATE	X(10)	DISPLAY	2013-05-10	
5	DB2-LASTCHANGED	X(26)	DISPLAY	2020-05-25-17.2...	
3	DB2-BROKERID-INT	S9(9)	BINARY	0	
77	IND-BROKERID	S9(4)	BINARY	0	
5	DB2-BROKERSREF	X(10)	DISPLAY		
77	IND-BROKERSREF	S9(4)	BINARY	0	
3	DB2-PAYMENT-INT	S9(9)	BINARY	0	
77	IND-PAYMENT	S9(4)	BINARY	0	
3	DB2-M-MAKE	X(15)	DISPLAY	TOYOTA	
3	DB2-M-MODEL	X(15)	DISPLAY	CAMRY	
3	DB2-M-VALUE-INT	S9(9)	BINARY	8500	
3	DB2-M-REGNUMBER	X(7)	DISPLAY	ABCDL0L	
3	DB2-M-COLOUR	X(8)	DISPLAY	SILVER	
3	DB2-M-CC-SINT	S9(4)	BINARY	0	
3	DB2-M-MANUFAC...	X(10)	DISPLAY	1998-08-19	
3	DB2-M-PREMIUM-INT	S9(9)	BINARY	450	
2	DB2-M-ACCIDENT	S9(2)	BINARY	0	
3	DB2-VIN	X(20)	DISPLAY		
	WHERE				
	SOLCA				
EXEC SQL SELECT INTO [POLICY,COMMERCIAL]					
> Record:Record1					
EXEC SQL SELECT INTO [POLICY,COMMERCIAL]					
> Record:Record1					
EXEC SQL OPEN [Cust_Cursor,Zip_Cursor,Customer]					
> Record:Record1					
EXEC SQL CLOSE [Cust_Cursor,Zip_Cursor,Customer]					
> Record:Record1					
EXEC SQL FETCH [Cust_Cursor,Zip_Cursor,Customer]					
> Record:Record1					
EXEC SQL SELECT INTO [POLICY,CLAIM]					
> Record:Record1					
EXEC SQL FETCH [CusClaim_Cursor]					

Figure 4-15 Editing the test

By rebuilding their test and the modified application into their personal load libraries, Deb can run the unit test without the Db2 changes that are made by the DBA. By running the tests with code coverage, they can ensure that the newly made changes are covered by the tests. All these tests can be run before they commit the code.

Figure 4-16 shows a sample of the code coverage report.

The screenshot displays two windows side-by-side. On the left is the 'Run As Test Case' dialog from the Z/OS Automated Unit Testing Framework. It shows configuration details for a test case named 'SUMAN9.ZUNIT.BZUCHG' using member 'TLGPDB01'. On the right is a 'Code Coverage Report' window showing coverage statistics for 'LGIPDB01.LGIPDB01 cob' at 15% coverage, with 47 lines covered and 265 total lines. Below these windows is a large block of SQL code, likely the source being tested, which includes various SELECT statements and Db2 subsystem definitions.

Figure 4-16 Code coverage report

Stubbed integration or impact tests

Deb can also run their application integration tests by using the recordings that they made earlier. These tests can be run with ZVTP from their personal libraries to look at the impact of the new field. They need the field only in the API and not in the screen.

The tests show the mismatches from the values that are obtained from the defaulted **SELECT** statement variables, as shown in Figure 4-17.

The screenshot shows the IBM Z Virtual Test Platform interface. A 'Warnings found RC=4' message is displayed. The 'Statements' section shows a recorded SELECT statement with variables like GET_CONTAINER and SELECT. The 'Statement details' panel on the right shows the transaction, program, statement, and db2 subsystem information. The recorded and replayed data is also visible, showing binary data differences.

Figure 4-17 ZVTP shows mismatches after a program update

Deb can confirm whether these mismatches are expected (Figure 4-18 on page 47).

Figure 4-18 Determining whether mismatches are expected

As expected, there is no change in the values within the **SEND** statement that sends data to the screen, which confirms that the impact was prevented from showing on the 3270 screens.

After Deb is confident of their changes, they can check in the code and the unit tests into an SCM like Git (as shown in Figure 4-19). After the changes are checked in, they can be part of a continuous integration and continuous delivery (CI/CD) pipeline that takes the new changes, builds them, runs unit and ZVTP tests, and deploys the changes into a test environment.

Figure 4-19 Pipeline report

The flow in Figure 4-20 summarizes what was done by a developer to perform tests before deployment.

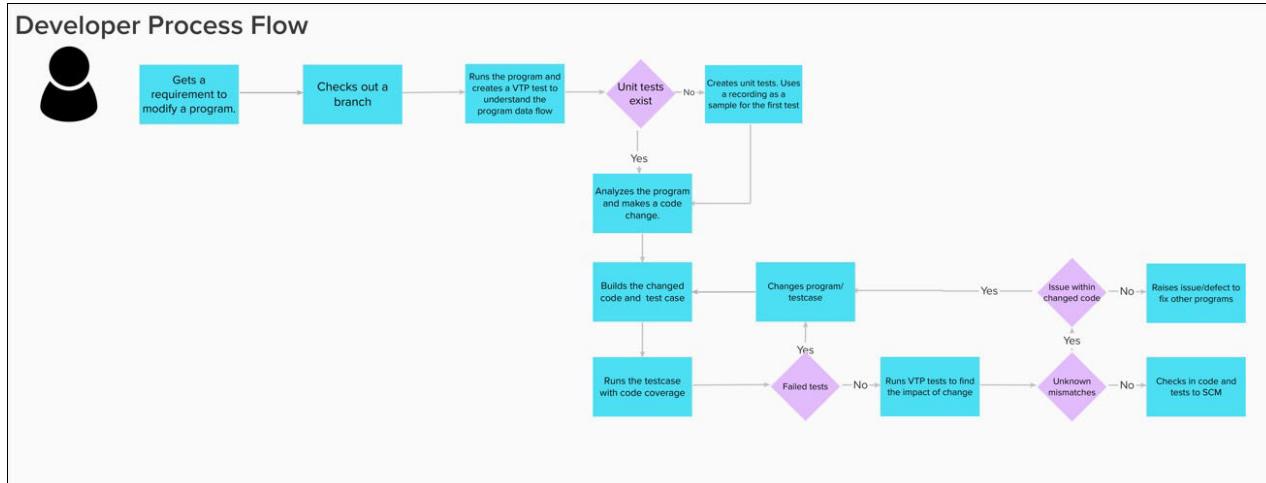


Figure 4-20 Developer process flow

4.2.4 Integration testing

Early tests help a developer to confirm that the changes that are made at a unit level and at a program-to-program integration level are correct. These tests also help to ensure that various boundary conditions are tested. However, this scenario does not eliminate the need for full integration tests after deployment that confirm that the values are updated correctly in the database.

There are tools that you can use to support this full end-to-end testing and drive testing at scale. Tools like Rational Test Workbench and the Galasa test framework are some of the tools that can be used to build advanced tests and then automate them.

Figure 4-21 on page 49 is an example of the Galasa architecture and how it can be used to drive tests end to end because the full application testing requires complex coordination between z/OS and distributed environments. Galasa tests can be driven from various front-end interfaces. Integration with the DevOps pipeline is key to achieving continuous testing.

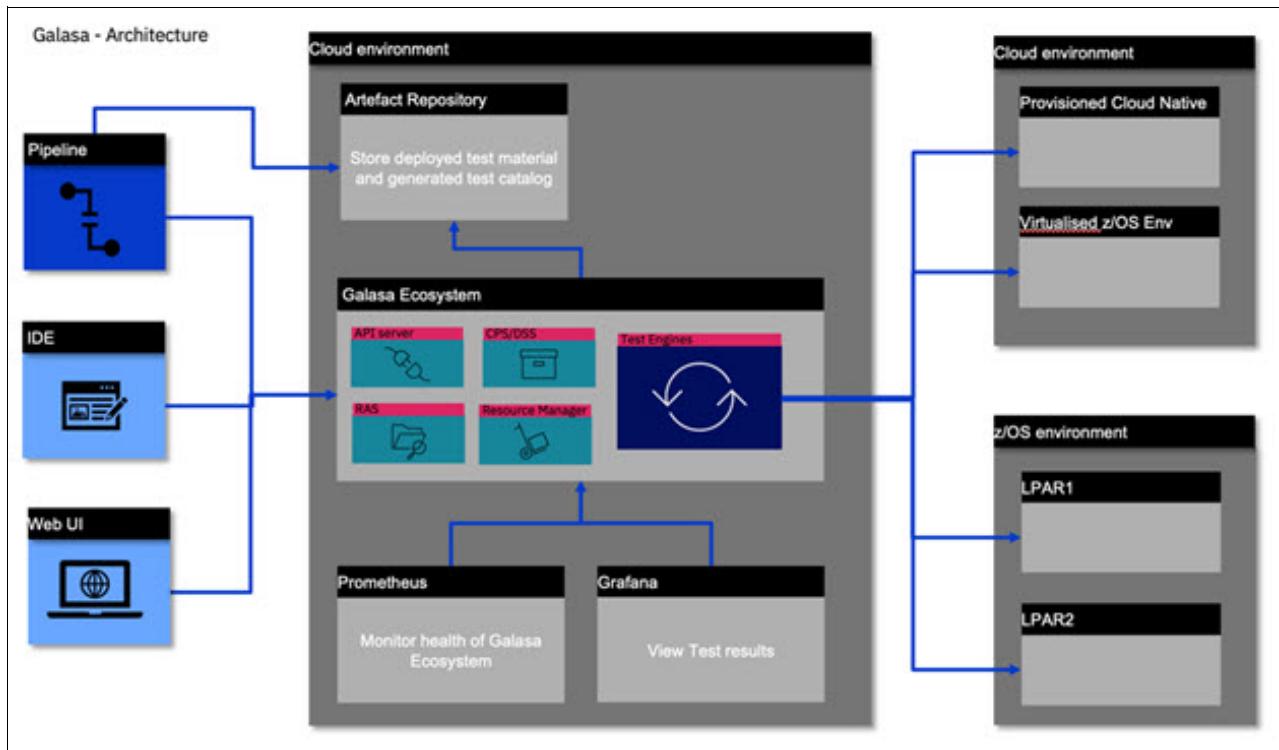


Figure 4-21 Galasa architecture

A tested program also can use the Galasa virtual test platform (VTP) manager to record VTP tests while the integration tests run so that VTP tests automatically are created from integration tests.

Figure 4-22 and Figure 4-23 on page 51 show an example of a Galasa test case.

```
38 import dev.galasa.zostsocommand.ZosTSOCommand;
39 import dev.galasa.zostsocommand.ZosTSOCommandException;
40 import dev.galasa.genapp.tests.TestUtils;
41
42 @Test
43 public class Ssp1recandpb {
44
45     @CoreManager
46     public ICoreManager coreManager;
47
48     @ZosImage(imageTag = "GENAPP")
49     public IZosImage image;
50
51     @Zos3270Terminal(imageTag = "GENAPP")
52     public ITerminal terminal;
53
54     @BasicGenApp
55     public IBasicGenApp genapp;
56
57     @ZosBatch(imageTag = "GENAPP")
58     public IZosBatch zosBatch;
59
60     @ZosBatchJobname(imageTag = "GENAPP")
61     public IZosBatchJobname zosBatchJobname;
62
63     @ZosTSOCommand(imageTag = "GENAPP")
64     public IZosTSOCommand tsoCommand;
65
66     @BundleResources
67     public IBundleResources resources;
68
69     @Logger
70     public Log logger;
71
72     @Test
73     public void deletePBFile() throws TestBundleResourceException, IOException, ZosTSOCommandException {
74
75         // Delete PB file
76         String tsoCommandString = "DELETE CICS.GENAPP.SSP1.PB";
77         String tsoResponse = tsoCommand.issueCommand(tsoCommandString);
78         logger.info("TSO response for delete of pb file=" + tsoResponse);
79     }
80
81     // Logging into the CICS-region with the provided CICS ID
82     @Test
83     public void login() throws InterruptedException, CoreManagerException, Zos3270Exception {
84         // Logon to the CICS Region
85         //terminal.waitForKeyboard().type("logon applid(" + genapp.getApplId() + "")").enter()
86         //               .waitForTextInField("Userid");
```

Figure 4-22 Example of a Galasa test case (1 of 2)

```

129     public void motorPolicyGenApp() throws InterruptedException, CoreManagerException, Zos3270Exception {
130         // Open the GenApp Motor Policy application
131         terminal.type("ssp1").enter().waitForKeyboard();
132
133         // Assert that the application menu is showing
134         assertThat(terminal.retrieveScreen()).containsOnlyOnce("General Insurance Motor Policy Menu");
135         assertThat(terminal.retrieveScreen()).containsOnlyOnce("1. Policy Inquiry      Policy Number");
136         terminal.positionCursorToFieldContaining("Issue date").tab().type("2020-01-01")
137             .positionCursorToFieldContaining("Cust Number").tab().type("0001000041")
138             .positionCursorToFieldContaining("Expiry date").tab().type("2021-01-01")
139             .positionCursorToFieldContaining("Car Make").tab().type("Honda")
140             .positionCursorToFieldContaining("Car Model").tab().type("CRV")
141             .positionCursorToFieldContaining("Car Value").tab().type("10000")
142             .positionCursorToFieldContaining("Registration").tab().type("1")
143             .positionCursorToFieldContaining("Car Colour").tab().type("Blue").positionCursorToFieldContaining("CC")
144             .tab().type("11111").positionCursorToFieldContaining("Manufacture Date").tab().type("2020-01-01")
145             .positionCursorToFieldContaining("Accidents").tab().type("000001")
146             .positionCursorToFieldContaining("Policy Premium").tab().type("0000050")
147             .positionCursorToFieldContaining("Select Option").tab().type("2").enter().waitForKeyboard();
148         assertThat(terminal.retrieveScreen()).containsOnlyOnce("New Motor Policy Inserted");
149         terminal.positionCursorToFieldContaining("Select Option").tab().type("3").enter().waitForKeyboard();
150         assertThat(terminal.retrieveScreen()).containsOnlyOnce("Policy Deleted");
151         terminal.pf3().waitForKeyboard();
152         terminal.clear().waitForKeyboard();
153
154     }
155
156     @Test
157     public void stopRecording() throws InterruptedException, CoreManagerException, Zos3270Exception {
158         // Stop the recording
159         terminal.type("bzue").enter().waitForKeyboard();
160
161         // Assert that the recording is stopped
162         assertThat(terminal.retrieveScreen()).contains("RECORDING STOPPED");
163         terminal.clear().waitForKeyboard();
164     }
165
166     @Test
167     public void writeRecording() throws InterruptedException, CoreManagerException, Zos3270Exception {
168         // Stop the recording
169         terminal.type("bzuw BVTQ").enter().waitForKeyboard();
170
171         // Assert that records are written to the queue
172         assertThat(terminal.retrieveScreen()).contains("RECORDS WRITTEN TO QUEUE BVTQ");
173         terminal.clear().waitForKeyboard();
174     }
175
176     @Test

```

Figure 4-23 Example of a Galasa test case (2 of 2)



Best practices and recommendations

The goal of this paper is to provide development teams and leadership with the information that is needed to understand the importance of early testing, and that best practices and tools for agile testing can be applied to z/OS applications. Good tools for testing z/OS applications are fundamental, and the test tools that are shared in this paper help to address the myths around the inability to efficiently build and grow modern and agile testing practices.

Much information has been shared on testing practices and tools in this IBM Redbooks publication. In this chapter, we summarize those testing practices and tools and provide some recommendations about how to begin your agile testing journey. Additionally, we reemphasize some of the best practices.

5.1 Process and cultural recommendations while adopting test automation

Here are some testing best practices:

- ▶ Get commitment.

To achieve product delivery with higher quality, you must get organizational commitment and the required supporting investment to incorporate agile development.

- ▶ Understand the current challenges.

Evaluate and agree on what is causing delays to product delivery cycles and impacting quality, and plan to address these issues. Most organizations understand this best practice already and have the data to support it. Common challenges are as follows:

- Shared testing environments causing delays.
- Delays to get the test data refreshed.
- High re-creates when defects occur.
- High defect rates after code is deployed to a full integration test.
- Lack of skilled z/OS application developers and testers.

- ▶ Talk to the experts.

Getting help from experts who made the cultural transformation to adopt early testing as part of their agile is recommended. This task helps you understand identified successes and failures. Invest in a workshop with recognized experts that can provide education and guidance to address the most pressing challenges and take the next steps. Ensure that the experts can provide recommendations for tools.

- ▶ Decide on the toolset.

Understand and decide on the tools that can be used. Good tools are needed to run all phases of testing and automate testing.

- ▶ Keep it simple for the first project.

Keeping it simple is important. Agree on a defined Minimum Viable Product (MVP). For example, the project might be to improve unit testing, and z/OS Automated Unit Testing Framework (ZUnit) is being considered as the tool to use. Start by defining the application to be tested along with the tools, and narrow the tests to a few key use cases that can be demonstrated to the team.

- ▶ Develop plans to roll out.

The easy first project went well and the team likes the tools. Next, build a roadmap with plans for a roll-out across the organization. This roll-out might need to be done in increments, for example, starting with a single application team or even part of that team. Ensure that developers and testers can learn the tools. Use agile planning to manage.

- ▶ Do not give up. It is not easy to build and run an agile way of working. This task takes time, but your organization will benefit from more efficient product delivery and higher quality.

5.2 General best practices

Here are some general best practices:

- ▶ Treat tests and test cases as code.

Ensure that tests have versions like code and are stored in the same Source Control Management (SCM) repository as your code. The tests are maintained and updated as part of the build phase. As a best practice, store unit tests within separate folders in the same repository.

- ▶ Organize integration tests.

Decide on the organizational practice. Tests can be grouped by function or application. Integration tests can be in a separate repository if they span across applications. Early integration tests can be stored in the same repository as the source code. Choose the best method for your organization based primarily on teams, application affinities, and structures.

- ▶ Use tests as automated quality gates.

It is great to run tests within an integrated development environment (IDE), especially when you are writing the tests. However, ensure that the tests are part of a continuous integration and continuous delivery (CI/CD) pipeline, and that the results are exported to your place of audit and to the pipeline run. These tasks ensure the traceability that links every pull or merge request with the tests that run. The tests form the first level of gating mechanisms to decide whether code is ready to be deployed into a specific test environment.

- ▶ Go slow to run faster.

Do not try to create automated test cases for every program and every function within the application. A best practice is to start creating tests for every change. Ensure that initially that the team gets more time in the build phase to incorporate the idea of creating reusable unit tests. Similarly, for integration tests, identify the critical path and critical points of failure and ensure that they are automated first. This approach enables a good quality test bucket that is built over time.

5.3 Summary

Organizations worldwide are embracing DevOps to transform their end-end development process by establishing a highly automated CI/CD pipeline. Automation is not enough. Investment in good application-specific test tools also is required to enable best practices like shift-left testing, where unit and application integration testing occur earlier in the pipeline process so that problems are discovered sooner.

You benefit significantly from reduced risk and costs, and can deliver confidently to production faster, whether you provide new capabilities or test fixes to those mission-critical applications.

Abbreviations and acronyms

ADDI	Application Discovery and Delivery Intelligence
ADFz	Application Delivery Foundation for z/OS
BMS	basic mapping support
CI/CD	continuous integration and continuous delivery
DBB	Dependency Based Build
DTR	IBM Dynamic Test Runner
EWM	Engineering Workflow Management
FMID	function modification identifier
GenApp	General Insurance Application
HLASM	High Level Assembler
HLQ	High-Level Qualifier
IBM	International Business Machines Corporation
IDE	integrated development environment
IDz	IBM Developer for z/OS
IDzEE	IBM Developer for z/OS Enterprise Edition
MVP	Minimum Viable Product
RSE	IBM Remote System Explorer
SCM	Source Control Management
SIP	system initialization parameter
TTLS	Tunneled Transport Layer Security
VTP	virtual test platform
ZUnit	z/OS Automated Unit Testing Framework
ZVTP	IBM Z Virtual Test Platform



REDP-5683-00

ISBN 0738460893

Printed in U.S.A.

Get connected

