ENTERPRISE RESOURCE PLANNING METAMODEL TEST PATTERN

A Paper
Submitted to the Graduate Faculty
of the
North Dakota State University
of Agriculture and Applied Science

By

Naveed Ahmed Syed

In Partial Fulfillment of the Requirements
for the Degree of
MASTER OF SCIENCE

Major Department:
Software Engineering

May 2014

Fargo, North Dakota

North Dakota State University
Graduate School

Title

Enterprise Resource Planning Metamodel Test Pattern

By

Naveed Ahmed Syed

The Supervisory Committee certifies that this ***disquisition*** complies with North Dakota State University's regulations and meets the accepted standards for the degree of

MASTER OF SCIENCE

SUPERVISORY COMMITTEE:

Dr. Kendall Nygard
Chair

Dr. Saeed Salem

Dr. Bakr Mourad Aly Ahmed

Approved:

16 April 2015	Dr. Brian Slator
Date	Department Chair

ABSTRACT

Enterprise Resource Planning (ERP) system is a complex distributed software solution that costs millions of dollars to implement. Upgrading an ERP metamodel subsystem introduces critical risks in the functionality of all the ERP applications that are part of an ERP system. It is therefore imperative to verify such critical updates to an ERP system thoroughly before they are released to manufacture. The situation is further exacerbated when the ERP system in context happens to be a legacy one without sufficient automated tests.

In order to facilitate effective and efficient implementation of ERP metamodel tests, a verification pattern is presented here. Application of this pattern has shown significant savings in project costs and testing effort, increase in coverage, reliability and reusability of test automation to verify different ERP sub-products (for example UI portal, web services), as well as significant increase in productivity of test engineers.

ACKNOWLEDGMENTS

I would like to first thank and express deep appreciation and gratitude to Dr. Kendall Nygard for his continued support, help, and direction. My sincere thanks go to Dr. Saeed Salem, and Dr. Bakr Mourad Aly Ahmed for their advice as well as serving on the committee. I am also very grateful to my family and friends who supported and motivated me to accomplish this task.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF APPENDIX FIGURES

INTRODUCTION

Background

"*Testing proves the presence, not the absence of bugs*" – E.W.Dijkstra

According to a study commissioned by the National Institute of Standards and Technology software defects cost the U.S. economy an estimated $59.5 billion annually. Given virtually every business today depends on software, this cost is bound to grow unless it is checked. The study also states that more than half of this cost is borne by software users and the remaining by software developers and vendors. The major reason attributed to this cost is an inadequate software testing infrastructure. Companies worldwide are taking steps to address this issue resulting in a $13 billion software testing market, according to the Gartner Group.

Automated testing, among several techniques, is one of the popular techniques primarily due to its ability to execute test cases faster and without human intervention. It can be incredibly effective, giving more coverage and new visibility into the software under test. It also provides us with opportunities for testing in ways impractical or impossible for manual testing. The biggest advantage automation testing has over manual testing is less expensive test runs. There are many factors to be considered when planning for software test automation. Some of which are testing complexity, skills needed to design and implement automated tests, automation tools.

While it might be costly to be late to the market, it can be catastrophic to deliver a defective product especially in case of Enterprise Resource Planning (ERP) systems. Every subsequent version of ERP system is larger and more complex in terms of feature set; software failures in such a system can cost an enterprise millions of dollars. Choosing appropriate testing technique is therefore essential in order to not only increase quality bar of ERP releases but also prevent buildup of technical debt which may prove expensive in the long run.

ERP Metamodel Test Problem

ERP system architecture poses many challenges for testing:

1. It is a complex distributed system that integrates several application domains. Very few software engineers generally have expertise in all the application domains; additionally, entry level engineers may not even have deep expertise in one application domain.
2. It may cross-cut multiple technologies (for e.g. C, C++, Java (JEE), C# (.Net)). In a team, a software engineer is generally specialized to work in one technology stack.
3. It may comprise of multiple sub-products (for e.g. Client UI portal, webservices) that are dependent upon same ERP foundational subsystem. This increases the integration test effort.
4. Advancement of an ERP system may require upgrading or reengineering the foundational layer with newer technologies based on market demand of performance and features. This may have a significant impact on both its metamodel and metadata subsystems.
5. Current trend in ERP module development is that of using an appropriate domain-specific language which is then translated into platform specific model. Metamodel defines the abstract syntax of domain-specific modeling languages; and it plays a central role in ERP application development. Consequently, correcting errors in a metamodel can be expensive as dependent artifacts have to be adapted to the corrected metamodel.

In addition to the architectural challenges ERP software vendor companies in a race to market the product have aggressive time lines not only for development but also for verification of quality. As a result ERP foundational/core system's testing may only be limited to critical features and in some cases even those features are indirectly verified via higher level ERP applications.

When the schedule is tight and lack of automated tests, software engineers often focus on testing new features and not include regression testing. Another anti-pattern that is commonly observed in the industry is to verify foundational layers via user interface using record and playback tool; however it doesn't result in tests that are robust, maintainable or transferable as changes occur.

In the light of this challenging problem this paper provides a solution that addresses most of the concerns.

Objectives

The goal of this paper is to provide an ERP metamodel test pattern aimed in improving and simplifying testing effort to test changes in metamodel functionality without having to worry about (a) various underlying technologies and (b) multiple ERP system sub-products such as Client portal (user interface mode), non-UI mode (Application Programming Interface based) client components. In order to accomplish the goal, following are the objectives of the ERP Metamodel Test Pattern presented in this paper:

1. *Increase cumulative test coverage* to detect errors within ERP metamodel and metadata subsystems compared to a more common procedural approach of implementing test case or test script. The procedural approach focuses on a test case and then implements it in a linear fashion. This simplification makes it harder to extend a test step part of a large test case. By improving how a test case is structured, given the complexities of a complex hierarchical test sequence, I believe it is not only possible to extend intermediate test steps but also enable

construction of complex test scenarios by aggregation of related test steps there by increasing cumulative test coverage of a test automation.

2. *Provide repeatability and reusability to save time*. Many times during test case implementation testers have a need to test a unique scenario or quickly implement one-off scenario, unless there is a well-defined test framework to guide such needs, quite often testers resort to path of least resistance and that is to make copy of a test script to make necessary changes to support one-off test scenario.
3. *Improve test engineering team's productivity*. When test cases get complicated due to not only complexity in system functionality but also due to the system's network topology where in system components may be distributed on multiple nodes; there is a greater need to manage test data across process boundaries. A test framework that manages the complexities of interaction with distributed system components in a transparent way will boost a tester's productivity. Additionally, by following a consistent test pattern that streamlines how a test case can be implemented, so that it can be reused without any rewrite, will also greatly enhance not only productivity but also makes it easier for new team members to start implementing complex tests quickly.

These high level objectives can be distilled into high level requirements for design of an ERP Metamodel Test Framework.

Requirements

The ERP Metamodel Test Framework (EMTF) shall support the following high level requirements:

1. EMTF shall support implementation of tests for verification of ERP Metadata accessible via interfaces exposed by ERP Client in following modes:

    1.1 UI mode (client portal)

    1.2 Non-UI mode (direct API calls)

2. EMTF shall facilitate reuse of implemented tests to construct more complex test scenarios
3. EMTF shall provide a mechanism to implement test transparently such that a set of tests can be reused under different deployment topologies, specifically following topology must be supported:

    3.1 *1-tier*: Test harness and ERP System under test (SUT) is installed on one computer. Both the test harness and ERP System share process space; there isn't any inter-process communication.

        3.1.1 This situation applies when Non-UI mode ERP Client is loaded into Test Harness process in order to verify exposed ERP Metadata interfaces part of the ERP Client

    3.2 *2-tier*: Test harness and SUT do not share process space and may or may not be installed on same computer. This requires inter-process communication support by the test framework, for e.g. test harness verifying UI mode ERP Client.

Sections

The next section of the paper focuses on the high level design of the ERP Metamodel Test Framework comprising of various architectural views determining the scope of the framework. The following section details the design derived from the architecture covering core aspects of the framework. This section is followed by the analysis of the core ERP Metamodel test pattern. The subsequent section provides details regarding unit testing of this pattern. The last section of paper covers future work, list assumptions and constraints that were kept in mind while designing and finishing with the conclusion and references.

HIGH LEVEL DESIGN

"*By relieving the brain of all unnecessary work, a good notation sets it free to concentrate on more advanced problems, and in effect increases the mental power of the race.*"

– Alfred Whitehead, 1911

All aspects that are necessary to be part of a test framework for implementing a ERP Metamodel test automation with the objectives stated earlier, requires one to formalize the concepts so as to infer the commonalities and their variances in ERP domain. Essentially, commonalities determine a system's architecture or high level design and variances influence detail design. Unified Modeling Language is used in this paper as a way to visualize, specify, construct and document ERP Metamodel Test Framework's architectural blueprints as well as detail design.

Based on the requirements, following section covers conceptual architectural aspects that clarify the test framework's domain as well as scope.

ERP Metamodel Conceptual Model

The model in Figure 1, presents various concepts (represented by rectangular boxes) that are part of ERP domain, which influence the test framework design, as well as their relationships among them. For the purposes of clarity, a directional dotted line is used to indicate 'Is a subtype of' entity relationship'; for example UIMode Client entity is a subtype of ERPClient entity.

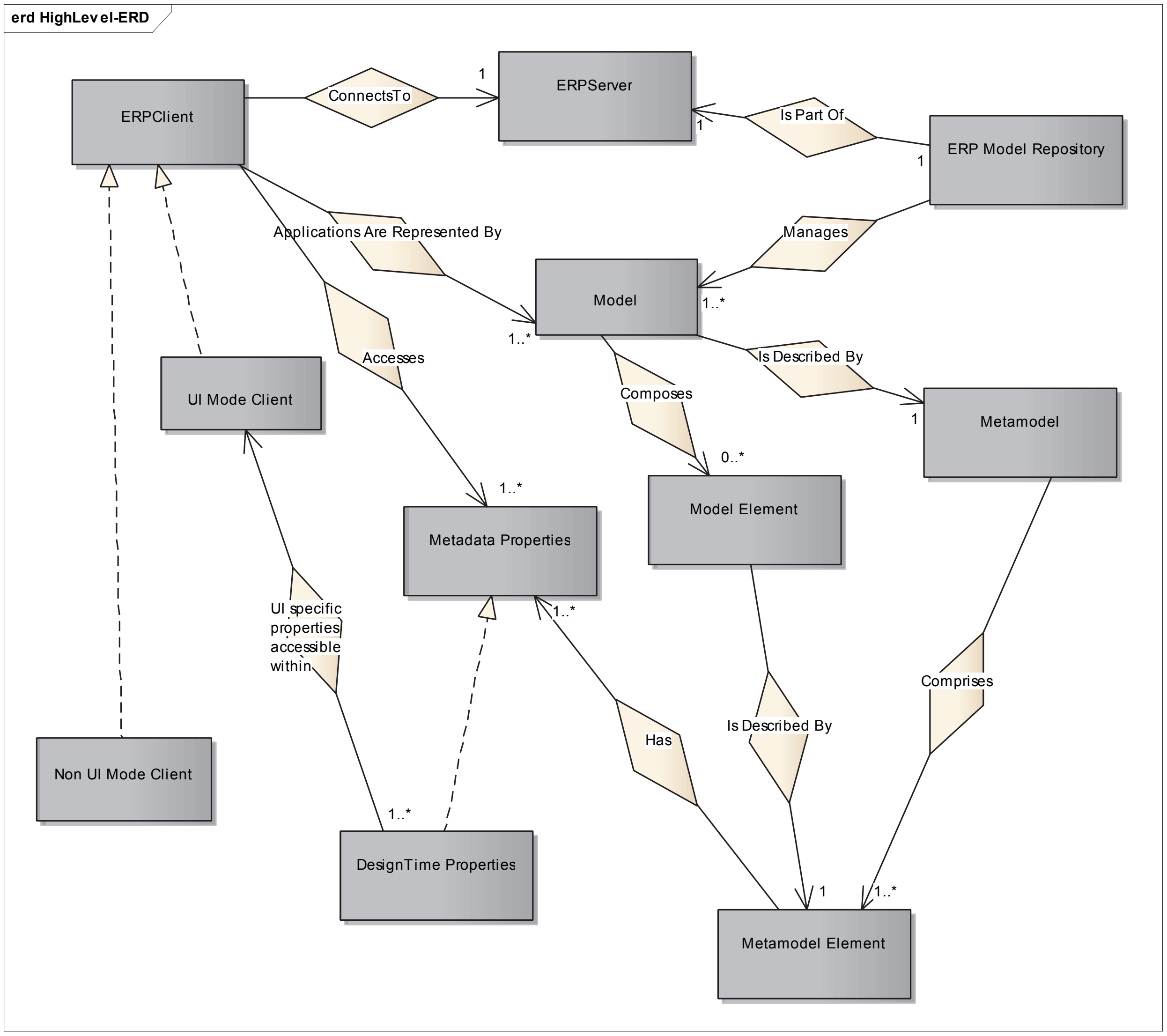


Figure 1: Entity Relationship Diagram Depicting the Context of an ERP Metamodel

ERPClient: It represents ERP Client that aggregates various ERP applications such as General Ledger, Account Receivables, Account Payables and so on. In addition to providing enterprise business functionality, it generally also exposes a modeling mechanism to model new ERP applications or customize existing ERP applications. ERPClient also exposes interfaces to access ERP Metadata.

UI Mode Client: In this mode the ERP Client provides a user interface for end users to interact with various ERP applications. Additionally, it also generally provides a modeling development environment to model various elements of an ERP application such as Table, Form, Form controls and so on. Latter aspect is utilized by the EMTF for verification of ERP metadata in this mode.

Non UI Mode Client: This mode lacks user interface and exposes a sub-set of UI Mode functionality via API – application programming interface – especially user interface model elements are not available. ERP systems support API access to ERP metadata via popular technologies such as Microsoft .Net, Microsoft COM (Component Object Model), Java. This increases the complexity of having to verify ERP metadata access via multiple mechanisms.

ERPServer: It plays the server role in Client-Server architecture. ERPClient communicates with it for all categories of ERP data and services. In this context it exposes interfaces for ERPClient to access ERP metadata of ERP models.

ERP Model Repository: It's part of the ERP Server and manages ERP Models. It also provides an interface for management of ERP Models, Model Elements.

Model: An ERP application is designed using Models and they comprise model elements such as Forms, Tables, Queries, Views, and so on. Basically it's a language for describing information domain.

Model Element: It is part of a model and has one to one correspondence with an ERP application concepts such Form, Tables, Form controls and so on.

Metamodel: It is a model that defines what can be expressed in valid models. Essentially it describes the structure of a model. It composes one or more Metamodel Elements. Also, it is an abstraction of Metadata. In other words, it is a specification of the content of a Model.

Metamodel Element: It is part of a Metamodel and it helps describes a Model Element; such as structure, relationships, behavior and constraints.

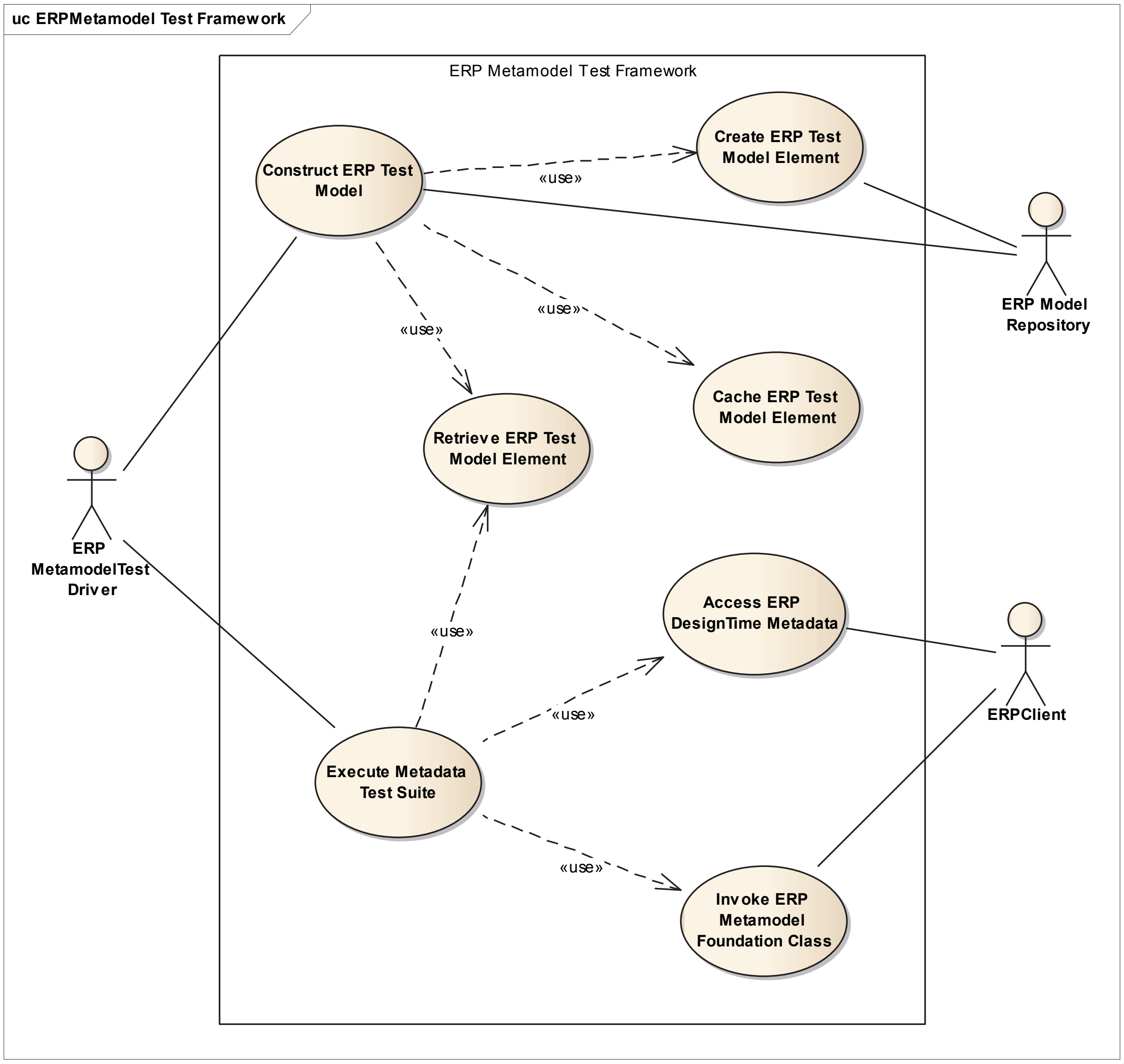
Metadata Properties: It is data which describes other data. ERP application domain model is modeled by metadata (for e.g. database schema, classes) which are parts of Metamodel. This data is provided by ERP Foundational Metamodel API.

DesignTime Properties: Metadata that is applicable when an application is being designed. Data describing Forms, Form controls and other user interface elements is only accessible from within ERPClient in UI Mode. Although, runtime metadata – event subscriptions metadata – verification and validation is outside the scope presented, the test pattern presented here can be extended to cover runtime metadata test scenarios.

Having covered the ERP domain as shown in Figure 1, following section on primary use cases presents the scope of the ERP Metamodel Test Pattern.

Primary Use Cases

Use Cases help describe a system from external viewpoint, representing objectives tester wants to achieve with a system. Given the context as mentioned earlier, the essential functionality (represented by ovals) that metadata test framework needs to support is shown in Figure 2. The stick figures represent external actors of the framework who either provide input or receive output or both.


uc ERPMetamodel Test Framework
ERP Metamodel Test Framework
Construct ERP Test Model
Create ERP Test Model Element
Retrieve ERP Test Model Element
Cache ERP Test Model Element
Access ERP DesignTime Metadata
Execute Metadata Test Suite
Invoke ERP Metamodel Foundation Class
«use»
«use»
«use»
«use»
«use»
«use»
ERP MetamodelTest Driver
ERP Model Repository
ERPClient
Figure 2: ERP Metamodel Test Framework System Boundary

Construct ERP Test Model: In order to verify and/or validate ERP metadata, a controlled test model is first constructed prior to running any test case. A model may either be created from scratch or an existing test model may be reused or extended. On successful construction the model is then persisted directly to ERP Model Repository.

Create ERP Test Model Element: A model composes model elements and hence construction of a test model results in calling use case to construct model elements. The state of a modeled element is used as expected test state (test oracle) in order to verify metadata returned by an ERP Client.

Cache ERP Test Model Element: Anytime a model element is either created or hydrated (deserialized from existing model), it is cached by the test framework to facilitate out-of-process communication in case of UI Mode ERPClient; where in a test driver is run in a different process to that of test cases, which is run in the context of ERP Client.

Retrieve ERP Test Model Element: A model element is retrieved to either populate the cache in case of out-of-process test scenario or to transparently access a model element's state during verification step.

Execute Metadata Test Suite: The pattern supports verification of two metadata aspects: (1) metadata exposed by ERP foundational Metamodel classes, (2) metadata that is applicable only during a model's design. Applying single responsibility principle, a Test Suite will focus on verifying one metamodel element and its associated metadata.

Access ERP Design Time Metadata: Metadata that is applicable only during a model's design phase is accessed by tests which is then compared with expected state for verification.

Invoke ERP Metamodel Foundation Class: Metadata exposed by ERP Foundational Metamodel classes is obtained by tests, calling respective interface and the response is then compared with expected state for verification.

Having covered ERP domain concepts and the scope of ERP Metamodel Test Framework (EMTF), next section focuses on the essential test aspects that directly shape the detail design later.

ERP Metamodel Test Assembly Conceptual Model

In the following figure, concepts with red rectangular border are external interfaces that test needs to integrate. This figure gives the static context of ERP Metamodel test framework as well as highlight concepts that must be supported to meet requirements stated earlier.

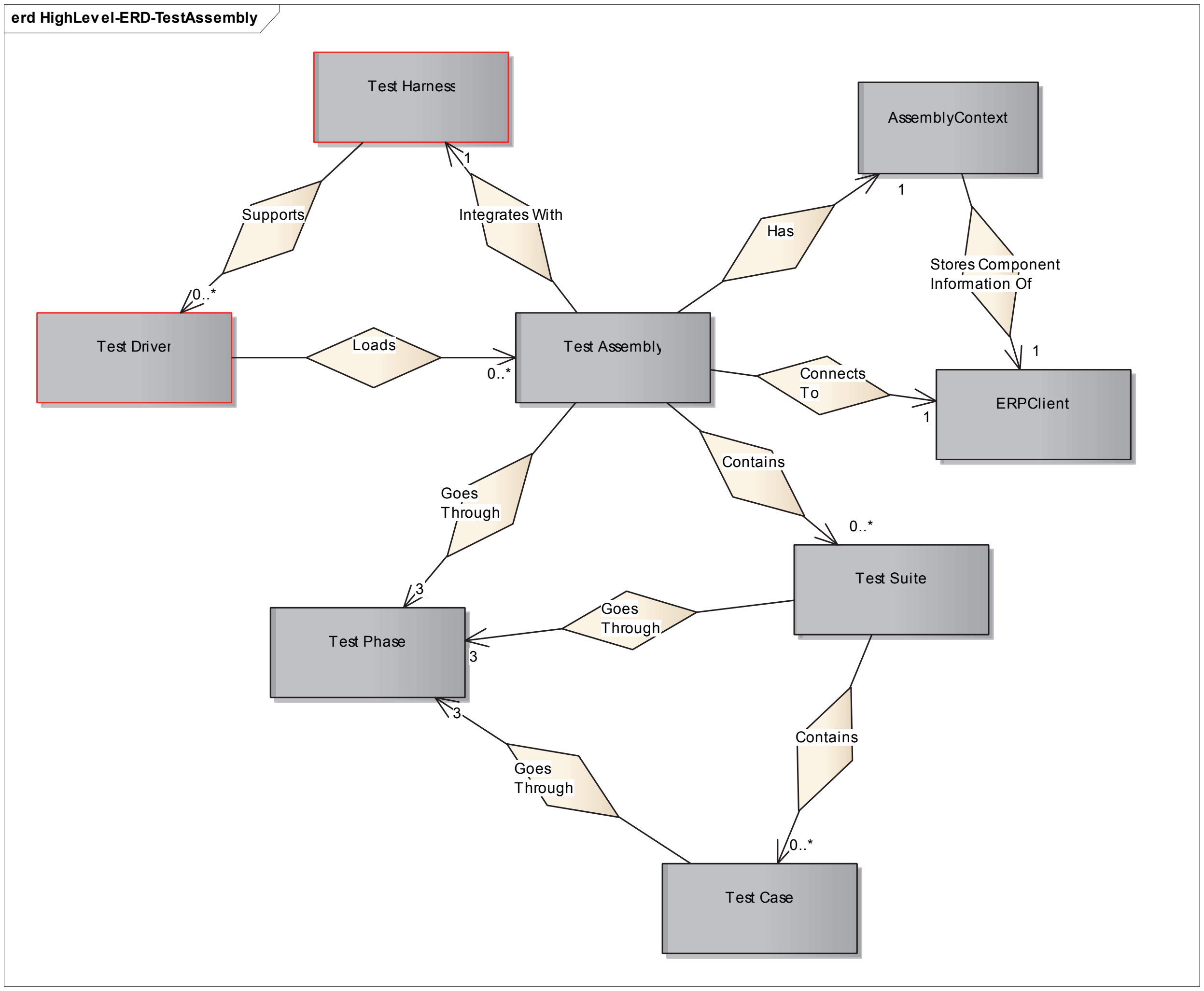


Figure 3: ERD Depicting Test Assembly Context

Test Harness: It is outside the scope of EMTF and is essentially a tool that executes tests packaged in a test library (Test Assembly in this context), gathers test results and generates test report at the end of an execution.

Test Driver: Part of a Test Harness that facilitates execution of library of tests across multiple Test Assemblies. EMTF integrates with such a test system via a set of design patterns to guide implementation of metadata tests.

Test Assembly: EMTF defines a contract for composition of Test Suites and is a unit of deployment. It has three distinct phases: setup, execution and cleanup.

Assembly Context: It maintains information regarding the System/Subsystem Under Test (SUT), i.e., ERP Client's Metamodel interfaces. This information comprises the kind of component (for e.g. UI Mode Client, Non UI Mode Client, and so on), component version and interface details for invocation to pass metadata requests into.

Test Suite: This abstraction is part of EMTF that defines the contract for implementation of metadata verification tests. Each Test Suite, contain a set of tests, that verify an aspect of an ERP Metamodel definition by accessing corresponding ERP Metadata. It is responsible for constructing an ERP Model Element. All related Test Suites are packaged into one Test Assembly that Test Driver loads in appropriate context for execution of tests. Test Suite concept is furthered analyzed in the section *ERP Metamodel Test Suite Conceptual Model*.

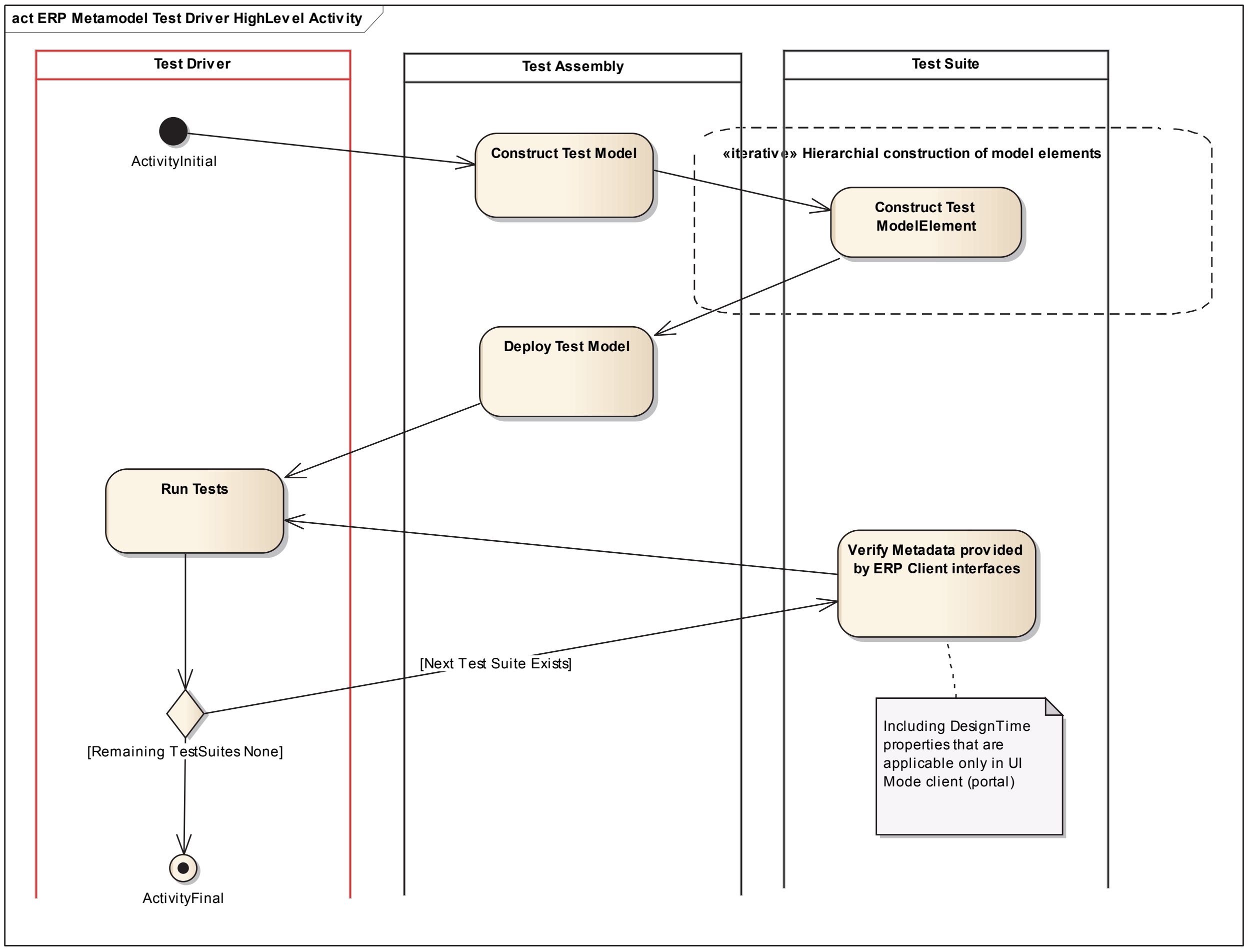
Test Case: It is a procedure that details steps to be executed in order to verify an aspect of Metamodel Element. It has dependency on a test oracle or expected state of system under test in order to verify the results observed during its execution. Test Case concept is furthered analyzed in the section *ERP Metamodel Test Case Conceptual Model*.

Test Phase: Every Test Assembly, Test Suite, Test Case has three distinct phases: setup (pre-execution), execution and cleanup (post-execution). Each of these phase defines what needs to be done so as to correctly observe output of a system under test (SUT) – ERP Metamodel Subsytem – given a controlled set of inputs.

While Figure 3 provided necessary concepts that test framework needs to support, next section covers essential dynamic behavior that must be supported by the test framework's detail design later.

Test Driver High Level Activity

Activity diagrams provide a simple and intuitive illustration of what a system execution flow looks like. These diagrams are sometimes enhanced using swim lanes to distinguish responsibilities as well as who is responsible for them. Following figure illustrates a very important and essential workflow that the EMTF design needs to support. The swim lane with red border represents an external interface, Test Driver that EMTF integrates with. Test Assembly and Test Suite responsibilities refine the requirements of EMTF.


act ERP Metamodel Test Driver HighLevel Activity
Test Driver
Test Assembly
Test Suite
ActivityInitial
Construct Test Model
«iterative» Hierarchial construction of model elements
Construct Test ModelElement
Deploy Test Model
Run Tests
Verify Metadata provided by ERP Client interfaces
[Next Test Suite Exists]
Including DesignTime properties that are applicable only in UI Mode client (portal)
[Remaining TestSuites None]
ActivityFinal
Figure 4: Test Driver High Level Activity

Test Assembly: It is responsible for providing a default valid model as well as integrates with ERP Model Repository for model management. Additionally, it contains one or more Test Suites.

Construct Test Model: EMTF defines a contract that a Test Suite needs to implement in order to construct an ERP Model Element during Test Assembly setup phase. Collection of all Test Suites together construct a whole Test Model. Construction of Test Model means either building a new ERP Model from scratch or reusing existing models.

Deploy Test Model: EMTF comprises functionality to deploy models built during Test Assembly setup phase to ERP Model Repository.

Run Tests: Tests that are contained within a Test Suite are run during Test Driver execution phase. This activity will continue as long as there exist a Test Suite to be run.

Verify Metadata: EMTF defines a contract that a Test Suite at a minimum needs to implement for verification of metadata obtained from an ERP Client.

Having covered both high level concepts and behavior of the test framework. Following section highlights key phases that test is taken through – directed by the test framework - during a test automation execution.

Test Phase Sequence

UML sequence diagram is a kind of interaction diagram that focuses on the message interchange between a number of lifelines. Each of the lifelines correspond to either an external actor, for e.g. Test Driver or concepts from ERD. The following figures clearly shows not only the messages exchanged between collaborators but also what messages trigger those communications. Additionally, it also shows how a test automation, during its lifetime, goes through three distinct phases of Setup, Execution and Cleanup.

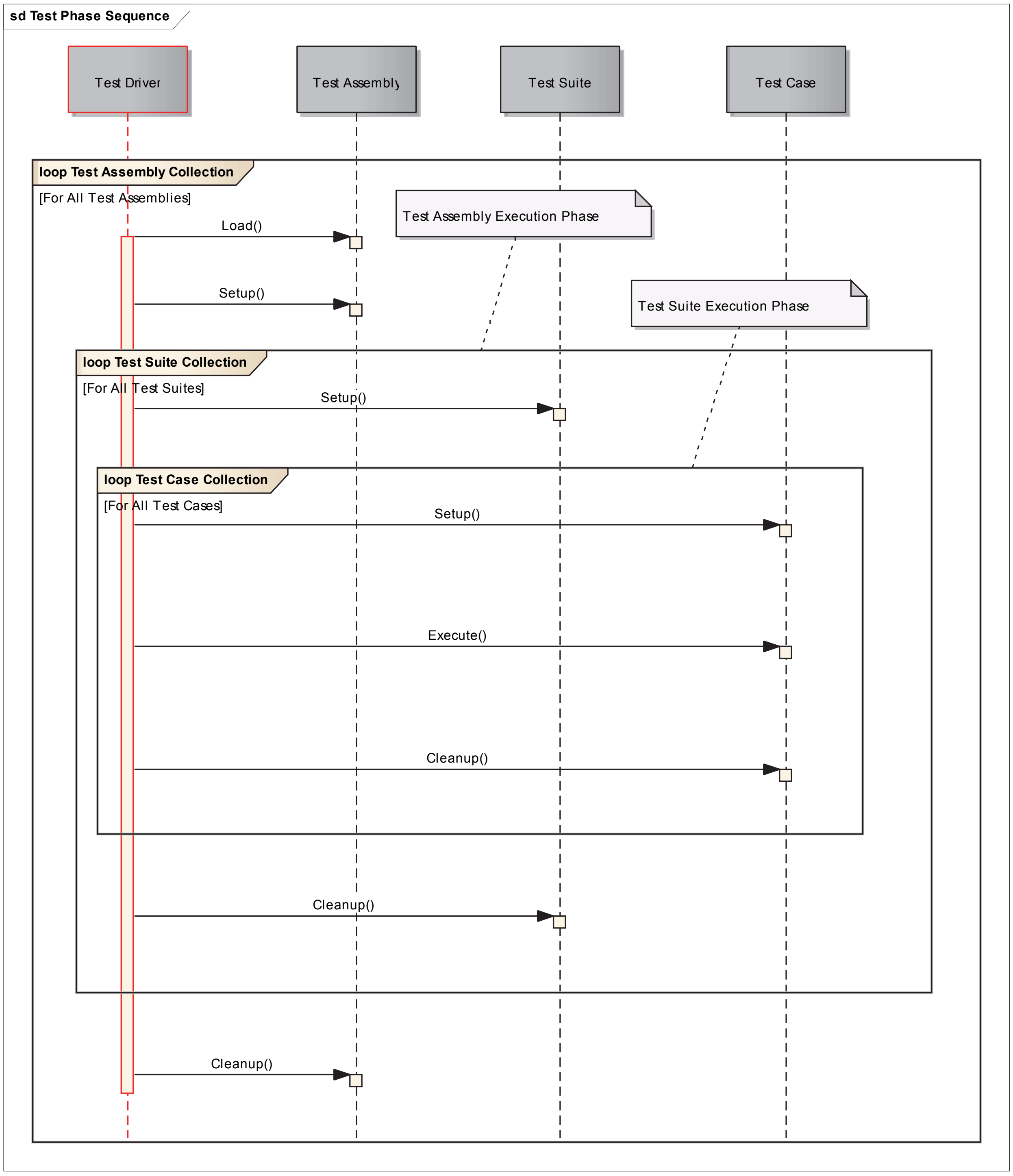


Figure 5: Test Phase Sequence Depicting Change of Phases

Test Assembly is essentially a container of Test Suites that allows an external test harness to execute ERP Metamodel test automation. TestSuite is the core abstraction that imposes constraints that guide to how an ERP Metamodel test is authored. Hence in next section TestSuite is further analyzed to identify necessary abstractions that are needed to be part of the test framework.

ERP Metamodel Test Suite Conceptual Model

The following ERD depicts concepts that are related to a Test Suite. Concepts with red rectangular border are part of ERP System specifically its foundational layer, outside the scope of the test framework. For the purposes of clarity, a directional dotted line is used to indicate ‘Is a subtype of’ entity relationship; for example RootLevel TestSuite entity is a subtype of TestSuite entity. Concepts introduced in this ERD will form the basis of detail design of ERP Metamodel TestSuite abstraction. Additionally, given the hierarchical nature of ERP Model Elements, ERP Metamodel TestSuites also have same hierarchical structure as each Test Suite is intended to verify only one Metamodel Element and composes other Test Suites to verify child Metamodel Elements.

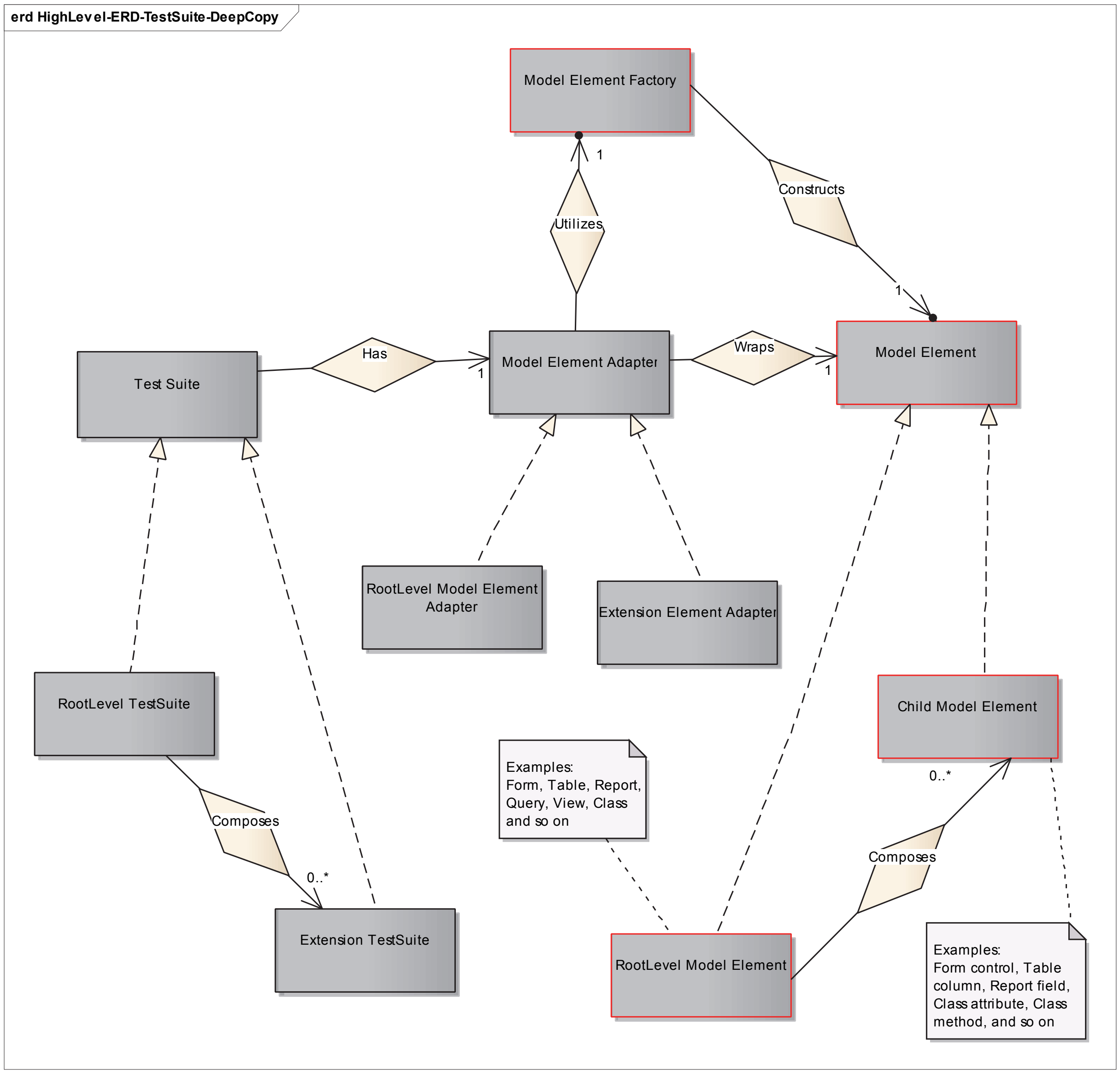

erd HighLevel-ERD-TestSuite-DeepCopy
Model Element Factory
1
Utilizes
Constructs
1
Test Suite
Has
1
Model Element Adapter
Wraps
1
Model Element
RootLevel Model Element Adapter
Extension Element Adapter
RootLevel TestSuite
Child Model Element
Examples:
Form, Table, Report, Query, View, Class and so on
0..*
Composes
0..*
Composes
Extension TestSuite
RootLevel Model Element
Examples:
Form control, Table column, Report field, Class attribute, Class method, and so on

Figure 6: Test Suite Perspective of an ERP Model Element

Root Level Model Element: This concept represents those Model Elements that do not need any parent for their existence; for example: Form, Table, Report, Query, View, Class and so on.

Child Model Element: This concept represents those Model Elements that do need a parent for their existence; for example: Form control, Table column, Report field, Class attribute, Class method and so on.

Model Element Factory: Provides an interface to construct an ERP Model Element. This abstraction is outside the scope of EMTF.

Model Element Adapter: This abstraction wraps an ERP Model Element and adds additional functionality such as caching for out-of-process communication. The cached Model Element as then used by Test Suite to retrieve expected test state during verification. EMTF provides two kinds of these adapters: Root Level Model Element Adapter and Extension Model Element Adapter.

Root Level Model Element Adapter: This abstraction is designed to wrap primarily RootLevel Model Element. It has a dependency on Root Level Model Element.

Extension Model Element Adapter: This abstraction is designed to wrap either a ChildLevel Model Element or RootLevel Model Element. When a Root Level Model Element is associated with another Root Level Model Element then the latter is referred to as an Extension of the former. For example a Form bound to Table, in this case both are Root Level Model Elements and latter is considered an Extension of the former.

Root Level Test Suite: This Test Suite has a Root Level Model Element Adapter that provides necessary expected test state to verify metadata retrieved corresponding to the Model Element.

Extension Test Suite: This Test Suite has an Extension Level Model Element Adapter that provides necessary expected test state to verify metadata retrieved corresponding to its Model Element.

The ERP Metamodel Test Suite concepts presented so far highlight hierarchical structure of Test Suites, as well as provide a mechanism to obtain expected Model Element state via an appropriate Adapter exposed via this test framework. In order to implement specific ERP

Metadata tests there is also a need to access actual or observable state of an ERP Model Element which is then compared with expected state. The following figure covers essential concepts that support metadata information retrieval.

ERP Metamodel Test Case Conceptual Model

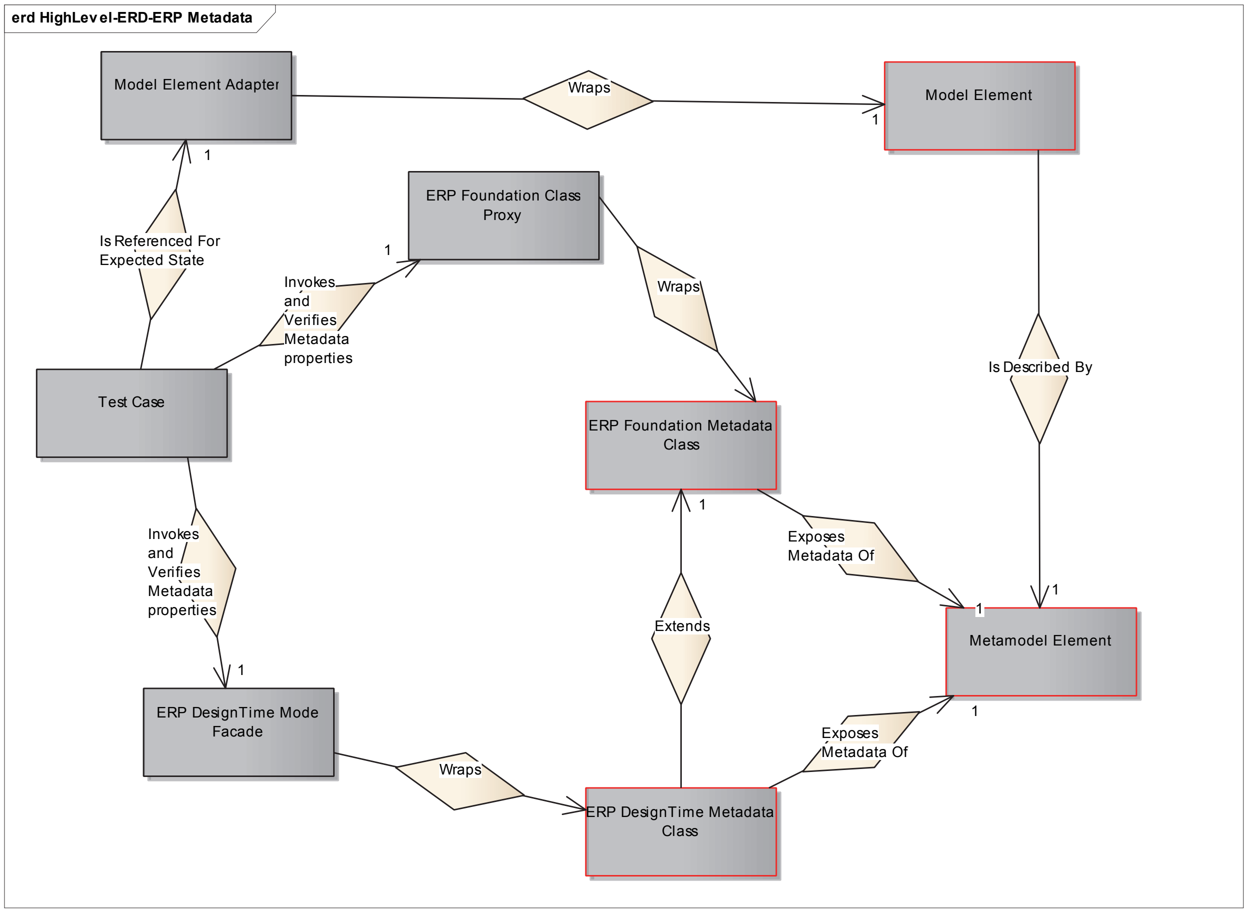

erd HighLevel-ERD-ERP Metadata
Model Element Adapter
Wraps
Model Element
1
ERP Foundation Class Proxy
Is Referenced For Expected State
Invokes and Verifies Metadata properties
Wraps
Is Described By
Test Case
ERP Foundation Metadata Class
Exposes Metadata Of
Invokes and Verifies Metadata properties
Extends
Metamodel Element
ERP DesignTime Mode Facade
Wraps
Exposes Metadata Of
ERP DesignTime Metadata Class

Figure 7: Test Case Related Concepts

ERP Foundation Metadata Class: Its part of an ERP System's foundation layer and provides an interface to access metadata for a given ERP Metamodel Element.

ERP Foundation Class Proxy: An ERP foundational class is exposed via an API. In order to abstract the complexities of invoking such API, ERP Test Framework that is being designed provides an abstraction in the form of a proxy for each ERP Metamodel Element. Tests implemented using these proxies are resilient to not only underlying changes to API technology but to also can be used to verify API version upgrades. Any impact due to changes in API interface is centralized within these proxies.

ERP DesignTime Metadata Class: Its part of an ERP System's foundation layer and provides an interface to access design time metadata for a given ERP Metamodel Element that is applicable in UI mode.

ERP DesignTime Mode Facade: Similar to the proxy concept mentioned earlier, façade enables a simple gateway to a complicated set of functionality of an ERP Metamodel Element. Accessing design time metadata in UI mode may require tests to leverage additional technologies to simulate UI interaction with ERP Client and observe metadata state. Usage of facades abstracts dependencies on various UI interaction technologies, thereby providing a consistent interface for test automations to use.

This section covered key concepts that are necessary to be supported by ERP Metamodel Test Framework in order to implement test automation for all ERP Metamodel Elements in a consistent way. The next chapter refines these concepts into a detail design.

DETAIL DESIGN

Methodology

"*Everything is vague to a degree you do not realize until you have tried to make it precise*" - Bertrand Russell

As a result of high level design analysis we have identified all the concepts or commonalities within ERP Metamodel domain. The next step now is to identify variances in those concepts so as to evolve detail design from the high level design.

The objective of detail design is to design a software test framework that will facilitate a consistent standard for implementing metamodel test automation and thereby making it easy to divide automated testing effort among engineers of varying skill levels. This is also intended to help a test engineer to focus primarily on test case implementation rather than ensuring the test automation is maintainable, reliable, flexible, efficient, portable, robust and usable.

This level of design was based on object oriented analysis and design principles leveraging well known *Gang-of-Four Design Patterns* [1]. These patterns are essentially recurring solutions to common problems of software design; and are an effective means of reusing existing expertise. Application of such patterns, where applicable, in concert results in desired software design structure. In addition to the well-known patterns the following guidelines were used:

1. Separate code that varies from the code that stays the same
2. Program to an interface and not to an implementation
3. Favor aggregation or composition over inheritance
4. Strive for loose coupling

5. Classes shall be open for extension but closed for modification
6. A class shall be assigned single responsibility
7. Design shall be flexible, robust and reusable

Test Framework Decomposition

Based on high level design, concepts are clustered into cohesive groups called as 'packages' as shown in Figure 8. All the packages are designed with the intent to cover necessary high level functionality and scope discussed in previous chapter. Utilities package is placeholder for functionality to handle a specific ERP environment and hence it will not be dealt any further in this paper.

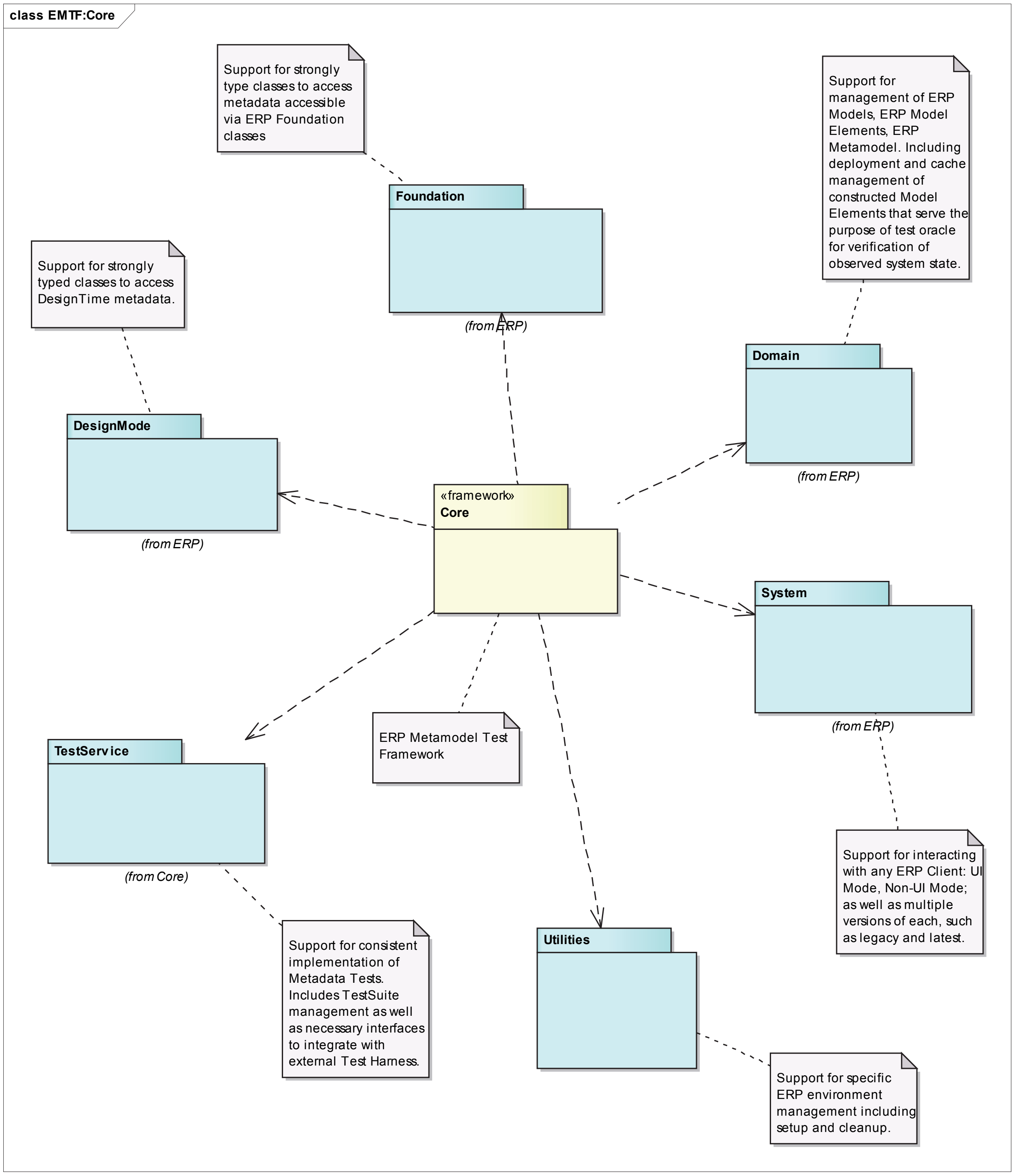

class EMTF:Core
Support for strongly type classes to access metadata accessible via ERP Foundation classes
Foundation
(from ERP)
Support for management of ERP Models, ERP Model Elements, ERP Metamodel. Including deployment and cache management of constructed Model Elements that serve the purpose of test oracle for verification of observed system state.
Domain
(from ERP)
Support for strongly typed classes to access DesignTime metadata.
DesignMode
(from ERP)
«framework» Core
System
(from ERP)
TestService
(from Core)
ERP Metamodel Test Framework
Support for interacting with any ERP Client: UI Mode, Non-UI Mode; as well as multiple versions of each, such as legacy and latest.
Support for consistent implementation of Metadata Tests. Includes TestSuite management as well as necessary interfaces to integrate with external Test Harness.
Utilities
Support for specific ERP environment management including setup and cleanup.

Figure 8: Test Framework Core Functionality Decomposition

Core: This software ‘package’ represents the core functionality part of the ERP Metamodel Test Framework (EMTF) and it composes six other packages as shown in the figure.

In this chapter each package is explored in the order of a test automation’s need based on Figure 4: System Package, Domain Package, Foundation Package, Design Mode Package and Test Services Package. System Package is first to be explored as follows.

System Package

This test framework package comprises of abstractions that enable instantiation and interaction with ERP Client components in a transparent way. Any interface changes in those external components affects only the abstractions in this package. Tests that depend on this package do not have to make any changes even when ERP Client version is changed, thereby making the tests dependable and resilient to SUT updates.

ERP System Class Diagram

In order to represent a static view of an object-oriented system UML Class Diagrams are used here for visualizing, describing and documenting different aspects of EMTF. It shows a collection of classes, interfaces, associations as well as relationships; besides describing a given class its attributes, operations and constraints imposed on the system. It is also known as a structural diagram.

In Figure 9 all tan color classes are part of core EMTF and the red and the green color classes represent variances of ERPClient supported by the test framework. ERPFactory provides method to create appropriate ERPClient instance based on the ComponentInfo parameter which specifies a type of the client to create. All the underlying objects from an ERPClient – ERPObject - are wrapped in an abstraction that is strongly typed and easier to work with.

There needs to be only instance of ERPClient during a test run, hence *Singleton Pattern* is used. Creation of ERPClient is done using *Factory Method Pattern* during TestAssembly Setup phase.

In this paper standard UML notation is used in all Class Diagrams. For example:

- *Realization Relationship*: IERPClient UML Interface is realized by ERPClient UML Class
- *Generalization Relationship*: ERPObject UML Class is generalization of ERPClient UML Class
- *Dependency Relationship*: ERPClient UML Class is dependent on ERPFactory UML Class; ERPClient uses ERPFactory but ERPClient does not contain an instance of ERPFactory as part of its own state. This relationship may be qualified by a '*stereotype*' for clarity purpose.
- *Association Relationship*: ComponentInfo UML Class uses and contains one instance each of UML Classes ComponentVersion and ComponentKind

Additionally, all UML structural diagrams are followed by corresponding one or more UML Sequence diagrams that clarify interaction of abstractions presented in the structural diagrams.

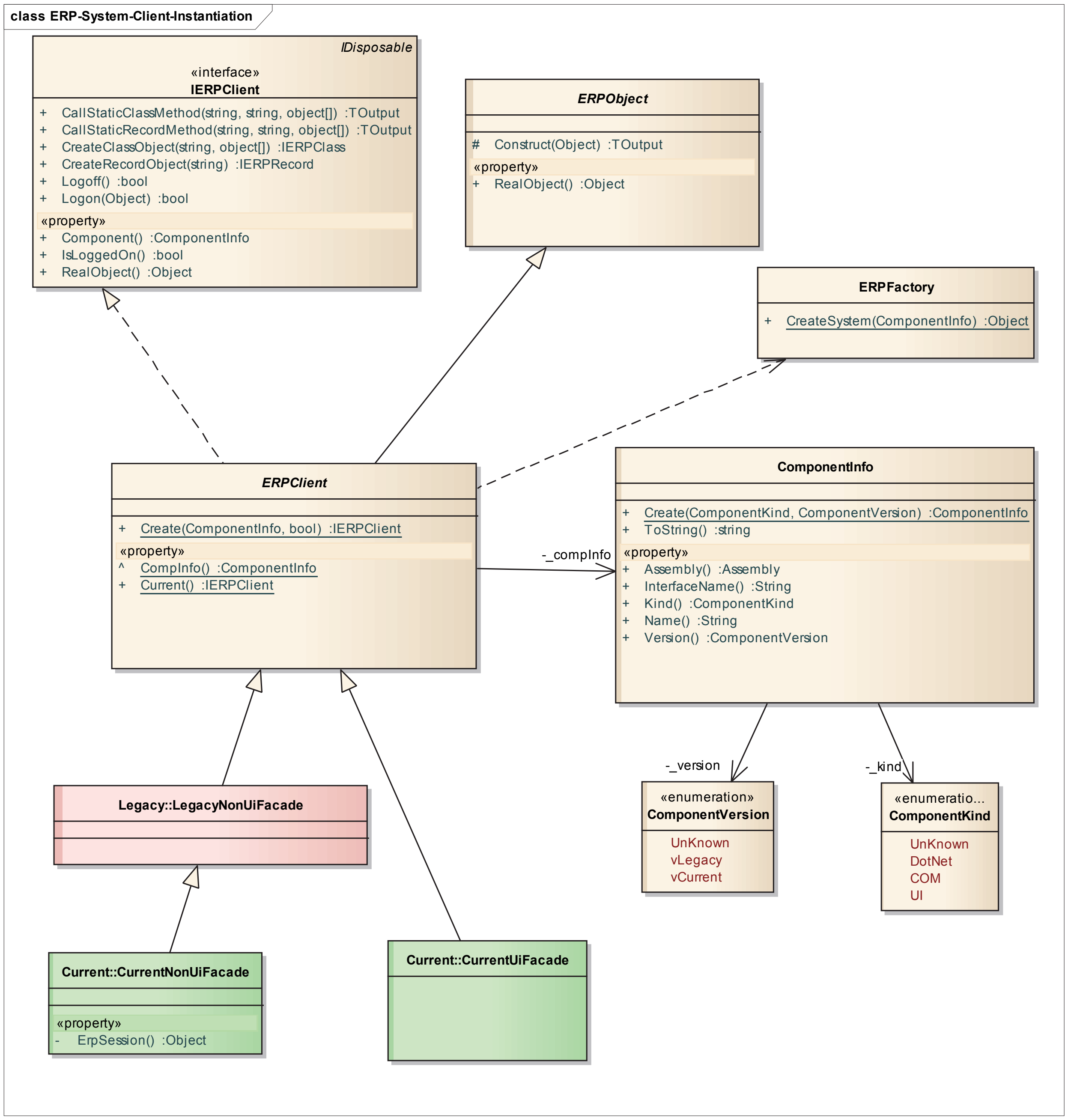
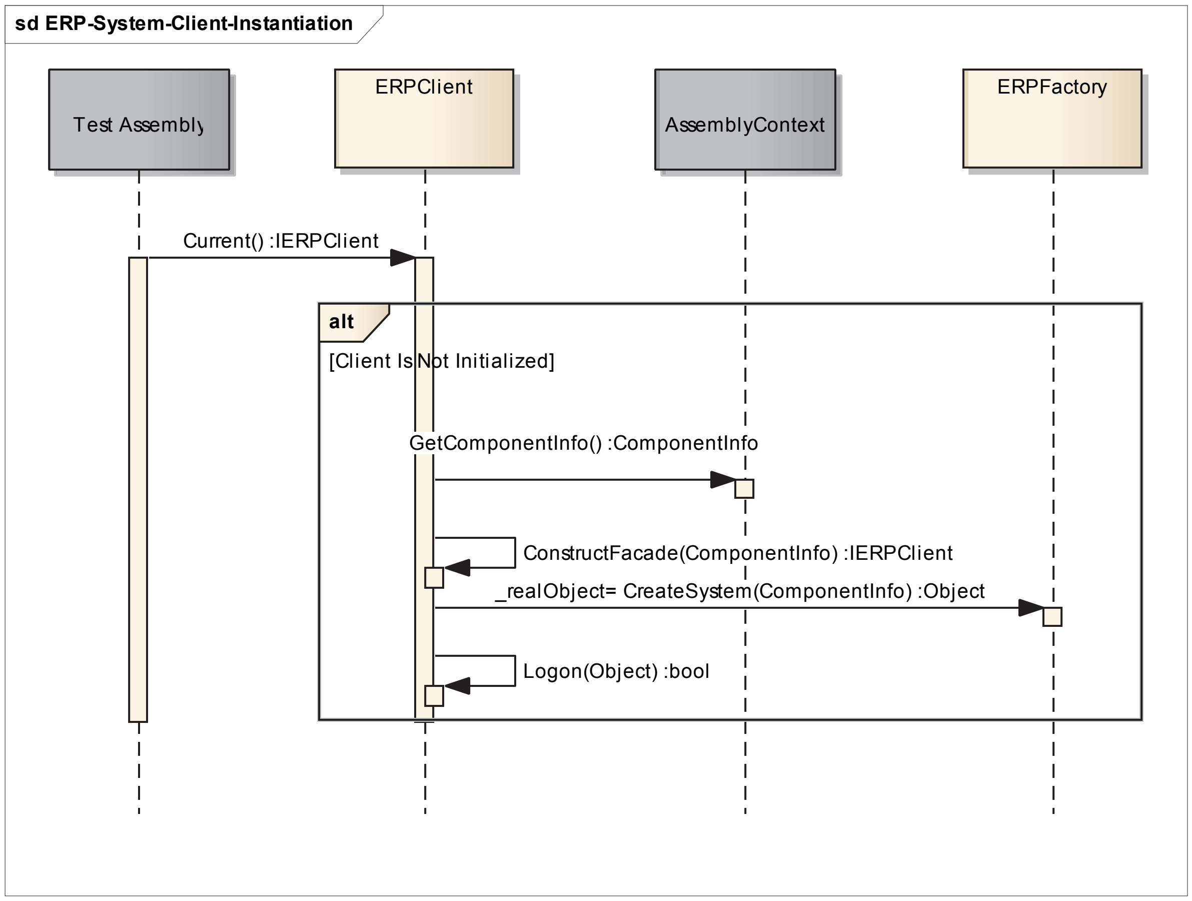

class ERP-System-Client-Instantiation
IDisposable
«interface»
IERPClient
+ CallStaticClassMethod(string, string, object[]) :TOutput
+ CallStaticRecordMethod(string, string, object[]) :TOutput
+ CreateClassObject(string, object[]) :IERPClass
+ CreateRecordObject(string) :IERPRecord
+ Logoff() :bool
+ Logon(Object) :bool
«property»
+ Component() :ComponentInfo
+ IsLoggedOn() :bool
+ RealObject() :Object
ERPObject
# Construct(Object) :TOutput
«property»
+ RealObject() :Object
ERPFactory
+ CreateSystem(ComponentInfo) :Object
ERPClient
+ Create(ComponentInfo, bool) :IERPClient
«property»
^ CompInfo() :ComponentInfo
+ Current() :IERPClient
-_compInfo
ComponentInfo
+ Create(ComponentKind, ComponentVersion) :ComponentInfo
+ ToString() :string
«property»
+ Assembly() :Assembly
+ InterfaceName() :String
+ Kind() :ComponentKind
+ Name() :String
+ Version() :ComponentVersion
-_version
-_kind
«enumeration»
ComponentVersion
UnKnown
vLegacy
vCurrent
«enumeratio...
ComponentKind
UnKnown
DotNet
COM
UI
Legacy::LegacyNonUiFacade
Current::CurrentNonUiFacade
«property»
- ErpSession() :Object
Current::CurrentUiFacade

Figure 9: Class Diagram of ERP System Abstractions

ERP System Initialization Sequence Diagram

While UML Class Diagram presented in previous section provided a static view of an object-oriented system, interaction among those abstractions that depict dynamic behaviour are shown below using UML Sequence Diagram. Such diagrams clarify collaboration among abstractions with the interaction starting near the top of the diagram and ending at the bottom.

Figure 10 shows instantiation of an ERPClient which starts with a TestAssembly requesting current singleton ERPClient, if it hasn't been initialized then ComponentInfo obtained from AssemblyContext is passed on to ERPFactory in order to construct IERPClient which is a *Façade* that facilitates easier access to an ERPClient's functionality.


sd ERP-System-Client-Instantiation
Test Assembly
ERPClient
AssemblyContext
ERPFactory
Current() :IERPClient
alt
[Client Is Not Initialized]
GetComponentInfo() :ComponentInfo
ConstructFacade(ComponentInfo) :IERPClient
_realObject= CreateSystem(ComponentInfo) :Object
Logon(Object) :bool
Figure 10: Sequence Diagram of an ERPClient Instantiation

System Package dealt with functionality supported by the test framework to interact with an ERP System. Next section on Domain Package focuses on management of ERP Models for test input.

Domain Package

This package comprises abstractions that support test ERP Model construction and deployment. An ERP Model's metadata is accessed, after deployment, via Proxies and Façades which are covered later. Concepts defined in Figure 6, are realized by this package.

ERP Model Construction Class Diagram

Given the hierarchical nature of Model concept and the complexity involved in building the model, *Builder Creational Pattern* is utilized to streamline construction of a Test Model. In this pattern a Director collaborates with Builders to build parts of a Model. IModelAdapter, part of Test Assembly, plays the role of a Director; and a TestSuite, that has one-to-one correlation with a given Metamodel Element, plays the role of Builder of an element. Due to composition of ExtensionTestSuites by either RootTestSuites or a container ExtensionTestSuite, such container TestSuites also play the role of a Director besides being a Builder themselves.

As shown in Figure 11, in order to support the construction of a Test Model, TestSuites implement IModelElementProvider interface, which delegates the responsibility of Model Element construction to IElementAdapter. Similar to a TestSuite, ElementAdapter has one-to-one correlation with a Metamodel Element. An ElementAdapter, using *Adapter Structural Pattern*, encapsulates necessary functionality to interact with ERP Model Repository for Model Element construction. Additionally, it implicitly caches a constructed Test Model Element so as to make it available during test verification step as expected test state.

A Test Model may also be constructed from an existing model in which case the same Builder pattern used during construction is also used here expect for one difference, and that is instead of requesting Builders to create new ModelElement, they are asked to 'Hydrate' –

populate in memory data structure which is done using *Prototype Creational Pattern* via Prototypes in the form of deserialized model elements.

By default, EMTF provides couple of baked Test Models: EmptyModel and ValidModel. EmptyModel doesn't have any elements in it. ValidModel is constructed based on a default set of TestSuites available for a test automation for specialization.

In Figure 11, turquoise colored classes – ClassTestSuite and MethodTestSuite - are examples of IRootModelElementProvider and IExtensionModelElementProvider respectively, which in turn delegate the responsibility of Model Element construction to IRootElementAdapter and IExtensionElementAdapter respectively.

In addition to standard representation of abstractions via UML Classes, 'generic' aspects of a class are also shown to clarify all other types a class might use. Such generic or unspecified types as *parameterized types* (for example 'T' as depicted in Figure 11) are supplied at point of use; when a parameterized type (T) is bound to a specific type it is denoted as 'T → *SpecificType*' in the UML class diagrams in this paper. For example, in Figure 11 shown below both RootTestSuite and ExtensionTestSuite have three generic parameters TestElementType, ERPClassType, DesignObjType where each of them must be comply with a design type contract as shown. TestElementType must be a Metamodel class, ERPClassType must be an ERP Foundation Class Proxy and DesignObjType must be a Façade that represents DesignMode Class. In case of a derived UML Class, for example ClassTestSuite, the binding of these generic types is also shown. TestSuite pattern is dealt in more detail in the section *ERP Metamodel Test Suite Pattern*.

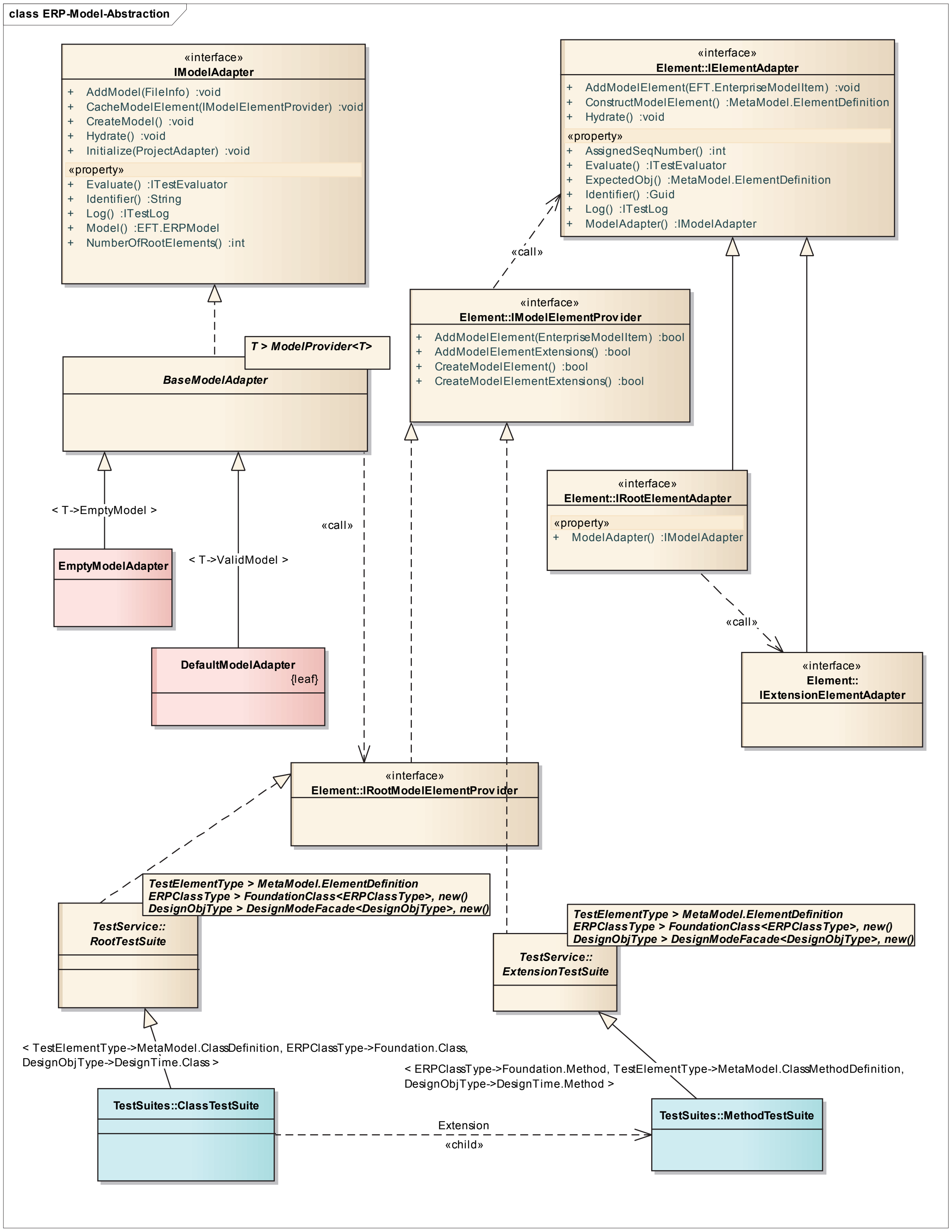

class ERP-Model-Abstraction
«interface»
IModelAdapter
+ AddModel(FileInfo) :void
+ CacheModelElement(IModelElementProvider) :void
+ CreateModel() :void
+ Hydrate() :void
+ Initialize(ProjectAdapter) :void
«property»
+ Evaluate() :ITestEvaluator
+ Identifier() :String
+ Log() :ITestLog
+ Model() :EFT.ERPModel
+ NumberOfRootElements() :int
«interface»
Element::IElementAdapter
+ AddModelElement(EFT.EnterpriseModelItem) :void
+ ConstructModelElement() :MetaModel.ElementDefinition
+ Hydrate() :void
«property»
+ AssignedSeqNumber() :int
+ Evaluate() :ITestEvaluator
+ ExpectedObj() :MetaModel.ElementDefinition
+ Identifier() :Guid
+ Log() :ITestLog
+ ModelAdapter() :IModelAdapter
«call»
«interface»
Element::IModelElementProvider
+ AddModelElement(EnterpriseModelItem) :bool
+ AddModelElementExtensions() :bool
+ CreateModelElement() :bool
+ CreateModelElementExtensions() :bool
T > ModelProvider<T>
BaseModelAdapter
< T->EmptyModel >
EmptyModelAdapter
< T->ValidModel >
DefaultModelAdapter
{leaf}
«call»
«interface»
Element::IRootElementAdapter
«property»
+ ModelAdapter() :IModelAdapter
«call»
«interface»
Element::
IExtensionElementAdapter
«interface»
Element::IRootModelElementProvider
TestElementType > MetaModel.ElementDefinition
ERPClassType > FoundationClass<ERPClassType>, new()
DesignObjType > DesignModeFacade<DesignObjType>, new()
TestService::
RootTestSuite
TestElementType > MetaModel.ElementDefinition
ERPClassType > FoundationClass<ERPClassType>, new()
DesignObjType > DesignModeFacade<DesignObjType>, new()
TestService::
ExtensionTestSuite
< TestElementType->MetaModel.ClassDefinition, ERPClassType->Foundation.Class,
DesignObjType->DesignTime.Class >
< ERPClassType->Foundation.Method, TestElementType->MetaModel.ClassMethodDefinition,
DesignObjType->DesignTime.Method >
TestSuites::ClassTestSuite
TestSuites::MethodTestSuite
Extension
«child»

Figure 11: Class Diagram of Abstractions Used for Model Construction

ERP Model Construction Sequence Diagram

The following UML Sequence Diagram presents the collaboration among model construction abstractions from previous section. While the diagram shows how a model element is constructed, hydration of model elements also follows exact same sequence but instead of create message, hydrate message is sent to collaborating class. Hierarchical tree structure is constructed in a recursive manner following a *Composite Structural Pattern*. As the child composites are created they are cached with their parent composite. Cached ModelElements are used as expected state during test verification phase. At the end of model construction it is deployed to ERP Model Repository.

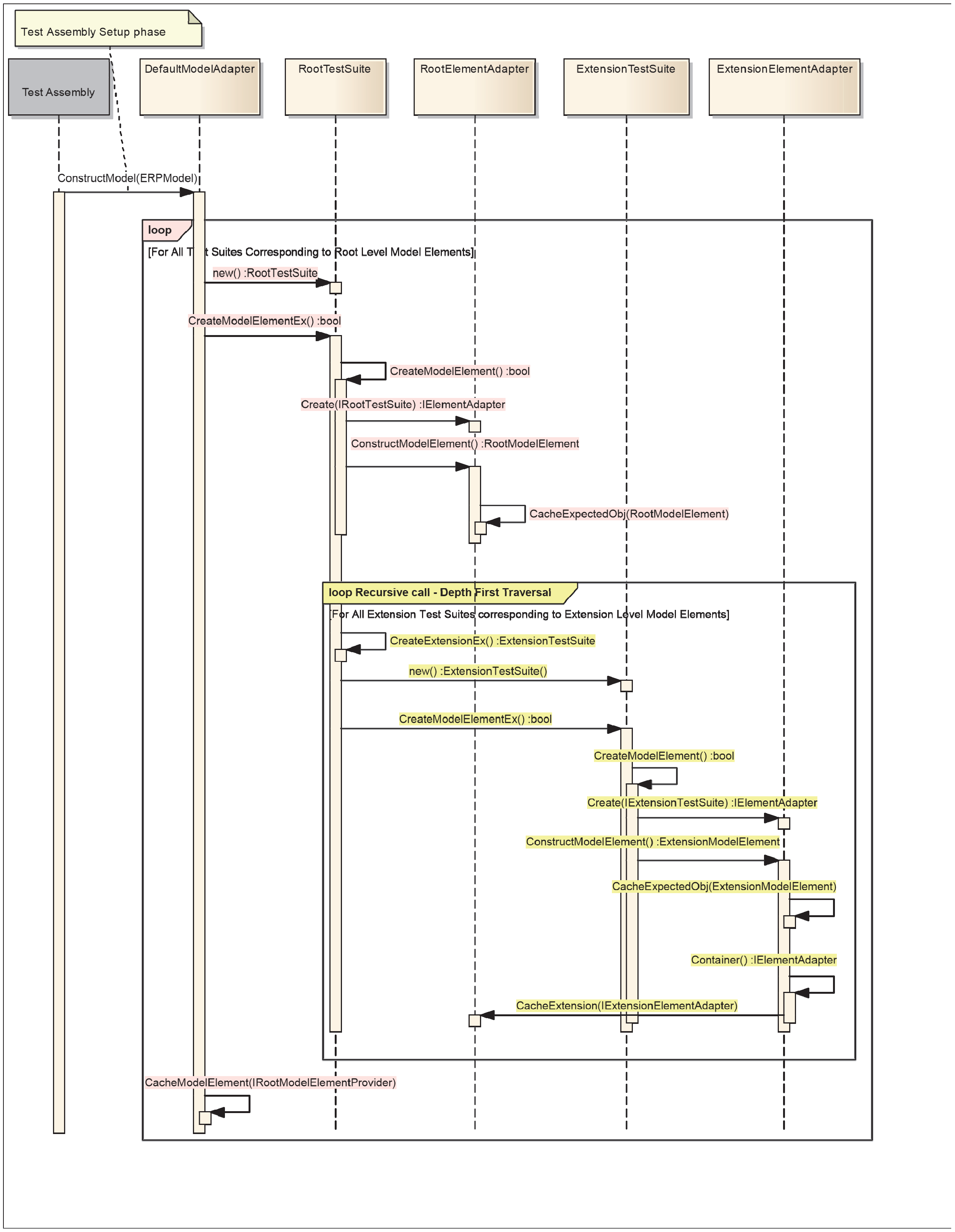

Test Assembly Setup phase
Test Assembly
DefaultModelAdapter
RootTestSuite
RootElementAdapter
ExtensionTestSuite
ExtensionElementAdapter
ConstructModel(ERPModel)
loop
[For All Test Suites Corresponding to Root Level Model Elements]
new() :RootTestSuite
CreateModelElementEx() :bool
CreateModelElement() :bool
Create(IRootTestSuite) :IElementAdapter
ConstructModelElement() :RootModelElement
CacheExpectedObj(RootModelElement)
loop Recursive call - Depth First Traversal
[For All Extension Test Suites corresponding to Extension Level Model Elements]
CreateExtensionEx() :ExtensionTestSuite
new() :ExtensionTestSuite()
CreateModelElementEx() :bool
CreateModelElement() :bool
Create(IExtensionTestSuite) :IElementAdapter
ConstructModelElement() :ExtensionModelElement
CacheExpectedObj(ExtensionModelElement)
Container() :IElementAdapter
CacheExtension(IExtensionElementAdapter)
CacheModelElement(IRootModelElementProvider)

Figure 12: Sequence Diagram Depicting Building of an ERP Test Model

Once an ERP Model is deployed, its metadata is observed by a test automation by accessing it in one of two ways: via API exposed by an ERP System's foundational layer or exposed by an ERP Client. Next section focuses on metadata access via the foundation layer.

Foundation Package

This package comprises necessary core abstractions that facilitate definition of strongly typed wrappers for ERP Foundation Metadata Classes as well as provide a consistent way to invoke those wrappers, as described in Figure 7.

ERP Foundation Class Proxy Class Diagram

ERP Foundation Metadata Classes may not be easily instantiated due to the fact that these reside on the different network node and may not be available on the network node that a TestAssembly resides on. In addition to that complexity, especially in case of legacy ERP systems one may need to invoke such classes in a late bound manner, meaning there are not strongly typed classes to easily access them. Anytime an ERP system were to be upgraded then one may have to rework all the invocations to appropriately handle new or upgraded loosely typed objects returned from the upgraded ERP system. To handle such complexities ERP Foundation Proxy Class pattern is provided below so as to encapsulate necessary plumbing necessary for such invocation for all ERP versions.

As shown in Figure 13 FoundationClass is the base proxy abstraction for all the strongly typed proxies of ERP Foundation Metadata Classes. The *Proxy Structural Pattern* followed here provides not only a consistent way to define all the strongly typed proxies but also hides the complexities present in invocation of underlying real ERP Foundation Metadata Class. The generic base proxy – FoundationClass – has a parameterized type that specifies all proxy subclasses must be derived from this base proxy as well as expose a public parameterless constructor.

Addition of a strongly typed proxy involves inheriting from generic base class FoundationClass and exposing properties that pertains to that specific FoundationClass. The base class makes it easy for a tester to create new instances of any proxy by exposing ‘Create’ factory methods. These ‘Create’ methods accept the singleton ERPClient instance and any applicable constructor parameters may also be passed into one of the overloads.

Depending on the ERPClient’s version the proxy base FoundationClass takes care of invoking appropriate version of underlying ERP Metadata Class. UML Classes in turquoise shown here are such proxy examples.

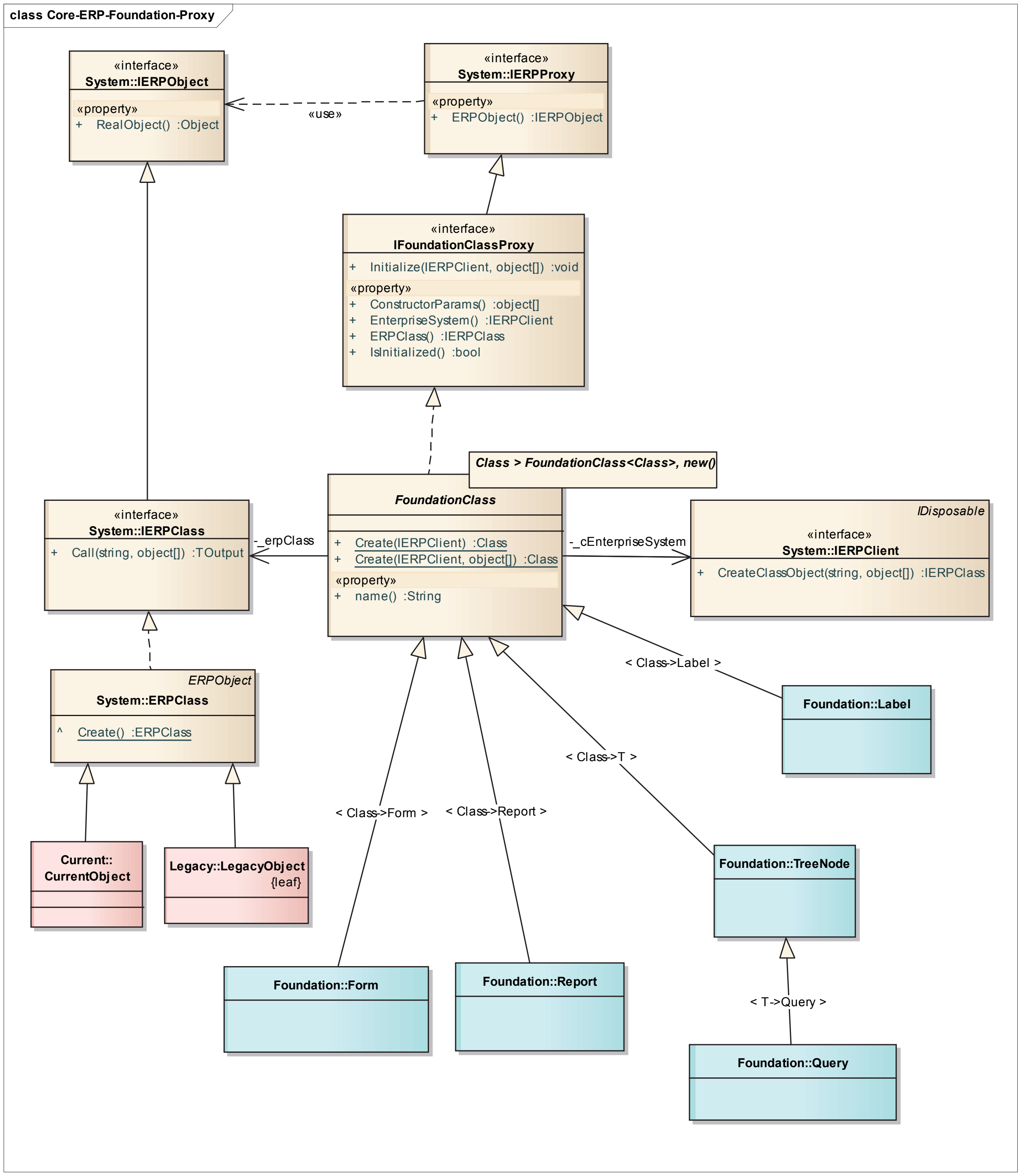
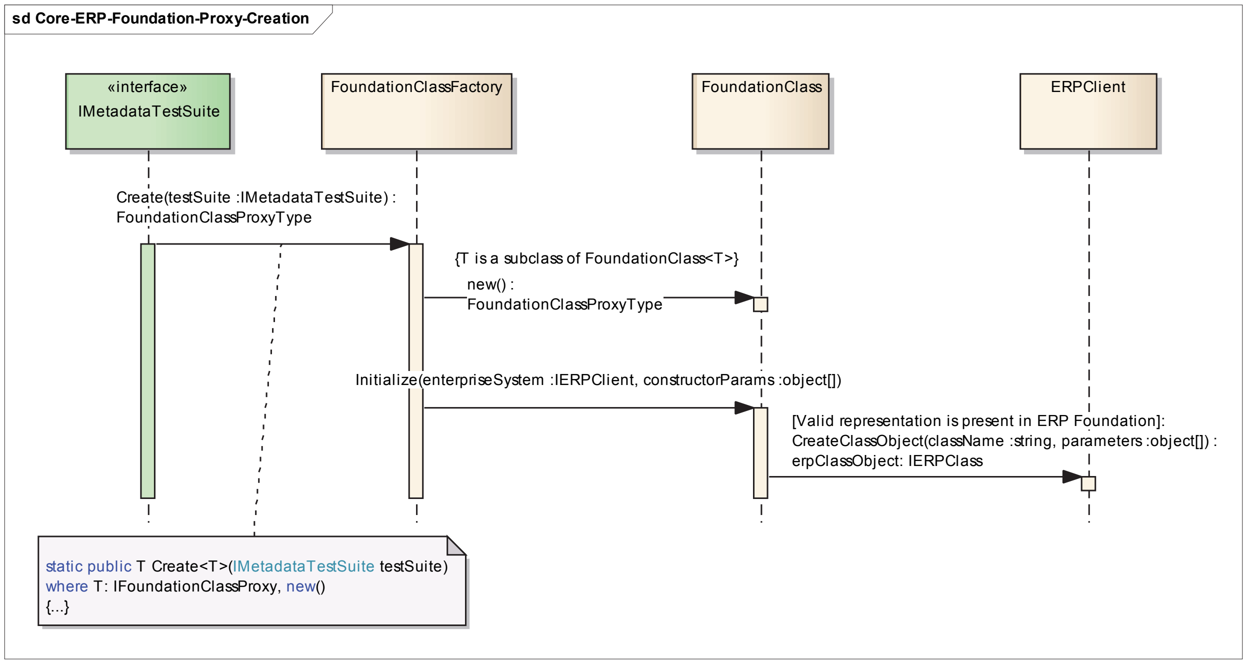


Figure 13: ERP Foundation Class Proxy Class Diagram

ERP Foundation Class Proxy Creation Sequence Diagram

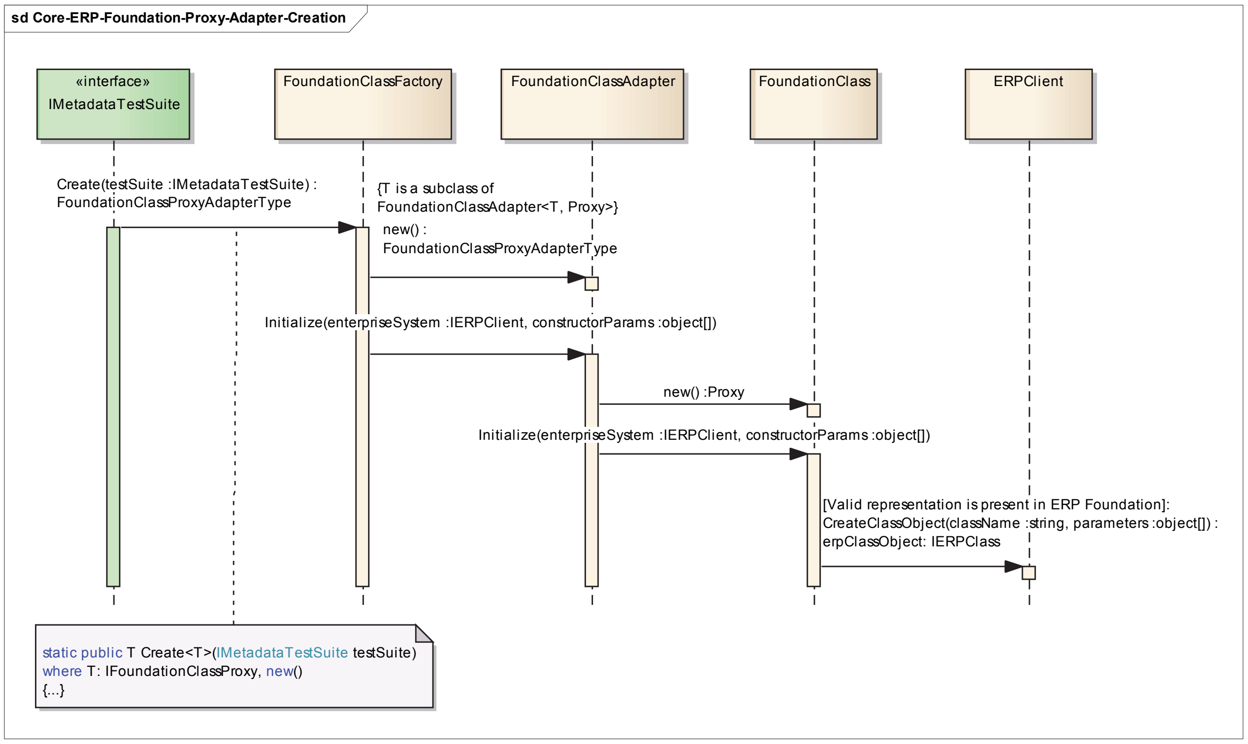
Construction of an ERP Foundation Class Proxy is shown in Figure 14. From testing perspective a Foundation Class Proxy is generally constructed by a corresponding TestSuite that validates a specific Metamodel Element. A TestSuite implements necessary parameters to construct corresponding FoundationClassProxy, FoundationClassFactory utilizes those parameters in contruction of a strongly typed FoundationClassProxy that wraps an ERP Foundation Class instance. The generic parameterized proxy factory method determines the type to instantiate based on the proxy type bound to the 'Create' method as shown in the figure below.


sd Core-ERP-Foundation-Proxy-Creation
«interface» IMetadataTestSuite
FoundationClassFactory
FoundationClass
ERPClient
Create(testSuite :IMetadataTestSuite) : FoundationClassProxyType
{T is a subclass of FoundationClass<T>}
new() : FoundationClassProxyType
Initialize(enterpriseSystem :IERPClient, constructorParams :object[])
[Valid representation is present in ERP Foundation]: CreateClassObject(className :string, parameters :object[]) : erpClassObject: IERPClass

```
static public T Create<T>(IMetadataTestSuite testSuite)
where T: IFoundationClassProxy, new()
{...}
```

Figure 14: ERP Foundation Class Proxy Creation Sequence Diagram

Creation of a Proxy is only possible when there is a corresponding real object for it to wrap. However, in cases where underlying FoundationClass is same for two different ERP Metamodel Elements, with just a difference in constructor parameters – for example Menu and MenuItem as shown in Figure 16, then an Adapter Proxy Pattern is used here. Basically, a FoundationClassAdapter wraps a FoundationClass Proxy to provide a meaningful as well as strongly-type definition for a TestSuite to work with. Figure 15 presents the sequence as to how such Adapter Proxies are created and initialized utilizing constructor parameters defined in a corresponding TestSuite; related structural diagram is provided in Figure 16.



sd Core-ERP-Foundation-Proxy-Adapter-Creation
«interface» IMetadataTestSuite
FoundationClassFactory
FoundationClassAdapter
FoundationClass
ERPClient
Create(testSuite :IMetadataTestSuite) : FoundationClassProxyAdapterType
{T is a subclass of FoundationClassAdapter<T, Proxy>}
new() : FoundationClassProxyAdapterType
Initialize(enterpriseSystem :IERPClient, constructorParams :object[])
new() :Proxy
Initialize(enterpriseSystem :IERPClient, constructorParams :object[])
[Valid representation is present in ERP Foundation]: CreateClassObject(className :string, parameters :object[]) : erpClassObject: IERPClass
static public T Create<T>(IMetadataTestSuite testSuite) where T: IFoundationClassProxy, new() {...}

Figure 15: ERP Foundation Class Proxy Adapter Creation Sequence Diagram

A tester's experience of adding new Foundation Class Adapter Proxies is similar to that of FoundationClass Proxies. The only difference in usage of Adapter Proxy factory method is that it expects a proxy class along with any applicable proxy construction parameters.

In cases where there simply isn't a way to access metadata via any ERP FoundationClass then a NullFoundationClass, a Null Object part of *Null Object Pattern* , provides a default behavior that seamlessly works with existing TestSuite collaboration.

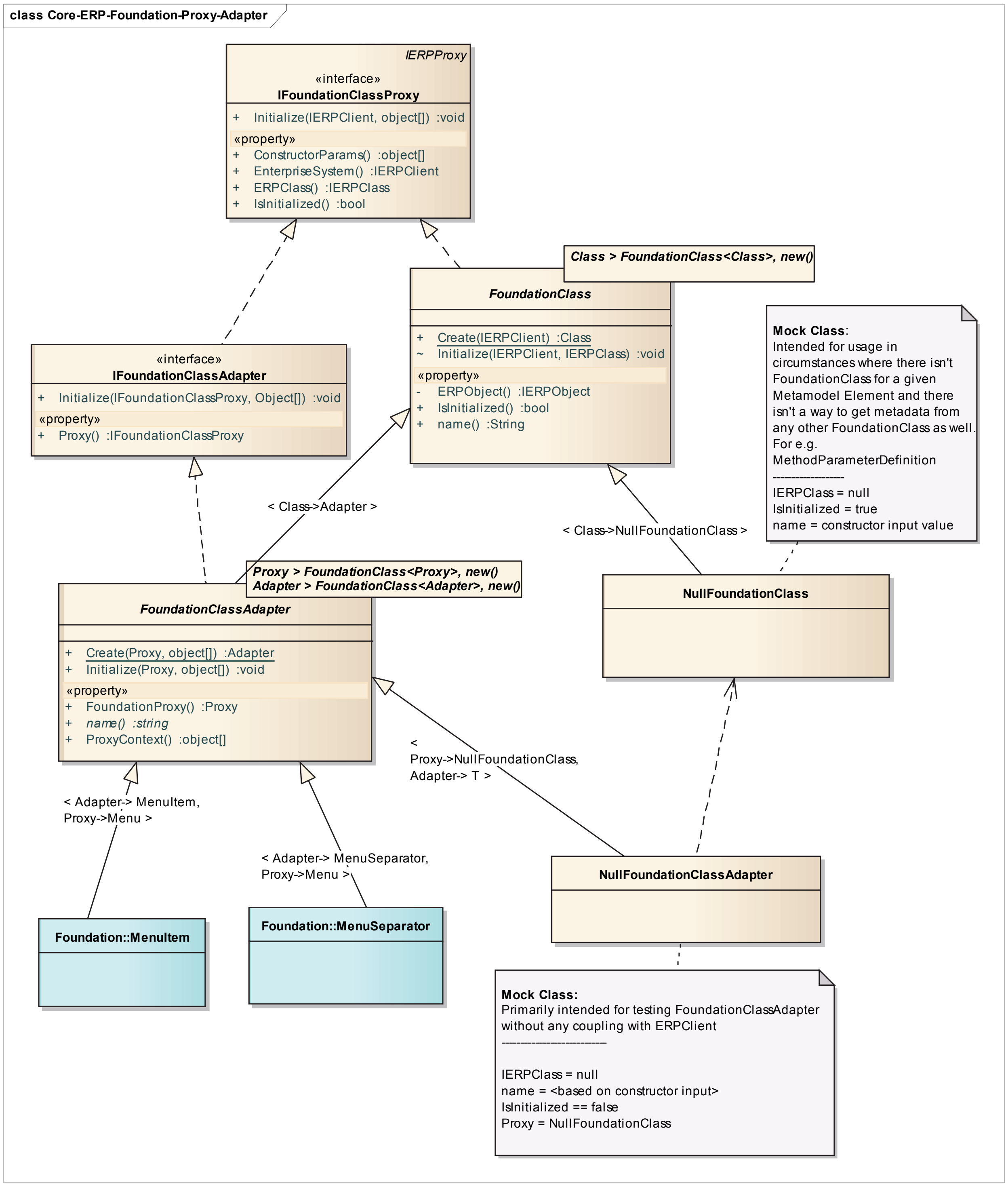

class Core-ERP-Foundation-Proxy-Adapter
IERPProxy
«interface»
IFoundationClassProxy
+ Initialize(IERPClient, object[]) :void
«property»
+ ConstructorParams() :object[]
+ EnterpriseSystem() :IERPClient
+ ERPClass() :IERPClass
+ IsInitialized() :bool
Class > FoundationClass<Class>, new()
FoundationClass
+ Create(IERPClient) :Class
~ Initialize(IERPClient, IERPClass) :void
«property»
- ERPObject() :IERPObject
+ IsInitialized() :bool
+ name() :String
«interface»
IFoundationClassAdapter
+ Initialize(IFoundationClassProxy, Object[]) :void
«property»
+ Proxy() :IFoundationClassProxy
Mock Class:
Intended for usage in circumstances where there isn't FoundationClass for a given Metamodel Element and there isn't a way to get metadata from any other FoundationClass as well. For e.g. MethodParameterDefinition
----
IERPClass = null
IsInitialized = true
name = constructor input value
< Class->Adapter >
< Class->NullFoundationClass >
Proxy > FoundationClass<Proxy>, new()
Adapter > FoundationClass<Adapter>, new()
FoundationClassAdapter
+ Create(Proxy, object[]) :Adapter
+ Initialize(Proxy, object[]) :void
«property»
+ FoundationProxy() :Proxy
+ name() :string
+ ProxyContext() :object[]
NullFoundationClass
< Proxy->NullFoundationClass, Adapter-> T >
< Adapter-> MenuItem, Proxy->Menu >
< Adapter-> MenuSeparator, Proxy->Menu >
NullFoundationClassAdapter
Foundation::MenuItem
Foundation::MenuSeparator
Mock Class:
Primarily intended for testing FoundationClassAdapter without any coupling with ERPClient
----
IERPClass = null
name = <based on constructor input>
IsInitialized == false
Proxy = NullFoundationClass

Figure 16: ERP Foundation Class Proxy Adapter

Having covered test framework mechanism to access ERP Metadata access via ERP Foundation Classes, next section covers mechanism for metadata access via another way that is exposed in user interface mode via an ERP Client.

Design Mode Package

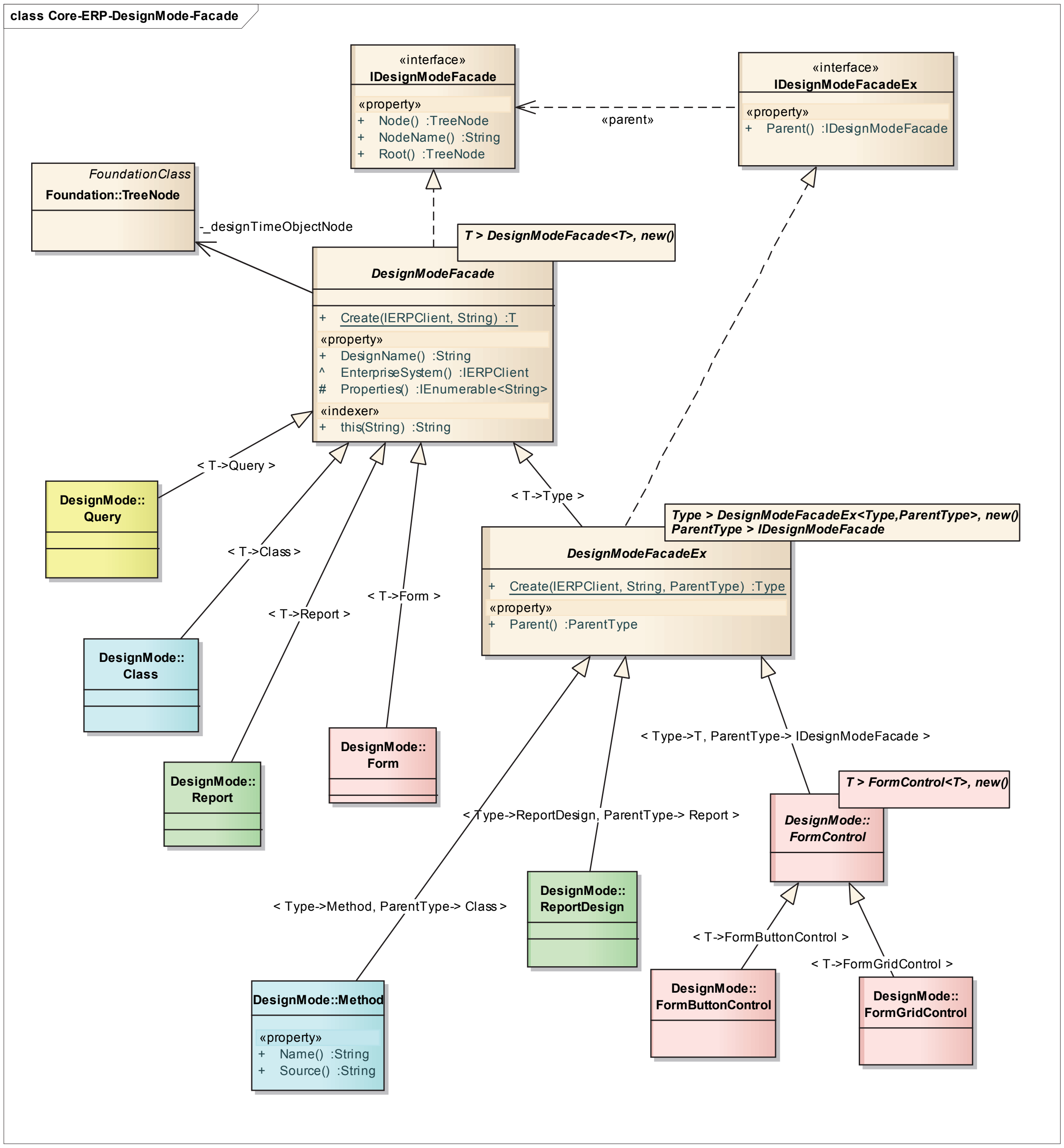
This package comprises core abstractions necessary to access metadata that is accessible within a UI Mode ERP Client as explained in Figure 1. Certain metadata is only available in UI Mode, for example a form layout, size and so on. These properties are generally set via an ERP specific design environment which is also accessed in an UI Mode, in some cases an UI Mode ERP Client also exposes this environment based on its process launch parameters.

An ERP design environment provides a way for developer to build ERP application by modeling forms, its associated form controls; tables and/or views that defines data the forms are bound to. Given the hierarchical nature of such an ERP Metamodel, EMTF models design time metadata corresponding to an ERP Metamodel Element as a TreeNode part of a hierarchical tree. Therefore, to obtain metadata accessible within such a design environment, a test need to traverse this conceptual tree. In order to make such traversal easier this package provides core abstractions that facilitate a consistent definition of strongly type classes to access ERP design time metadata. Additionally, these abstractions hide the complexity of invoking underlying ERP design mode classes much like that of ERP Foundation Classes.

ERP DesignTime Mode Façade Class Diagram

In Figure 17 abstract generic base class DesignModeFacade plays the role of a parent composite and abstract generic base class DesignModeFacadeEx plays the role of a child composite which requires the generic type parameter - ‘*ParentType*’ – be additionally bound;

and forms a part of *Composite Structural Pattern*. Similar to the core ERP Foundation Class Proxy pattern, the DesignModeFaçade pattern allows a tester to easily add new façades by inheriting from appropriate base classes, exposing façade specific strongly type properties and binding the generic type parameter '*T*'. Factory methods, defined in these base classes, are exposed for a consistent way to instantiate any design mode façade. Example DesignMode Façades are shown in yellow, turquoise, green and red colors.


class Core-ERP-DesignMode-Facade
«interface» IDesignModeFacade
«property»
+ Node() :TreeNode
+ NodeName() :String
+ Root() :TreeNode
«interface» IDesignModeFacadeEx
«property»
+ Parent() :IDesignModeFacade
«parent»
FoundationClass Foundation::TreeNode
-_designTimeObjectNode
T > DesignModeFacade<T>, new()
DesignModeFacade
+ Create(IERPClient, String) :T
«property»
+ DesignName() :String
^ EnterpriseSystem() :IERPClient
# Properties() :IEnumerable<String>
«indexer»
+ this(String) :String
< T->Query >
DesignMode:: Query
< T->Class >
DesignMode:: Class
< T->Report >
DesignMode:: Report
< T->Form >
DesignMode:: Form
< T->Type >
Type > DesignModeFacadeEx<Type,ParentType>, new()
ParentType > IDesignModeFacade
DesignModeFacadeEx
+ Create(IERPClient, String, ParentType) :Type
«property»
+ Parent() :ParentType
< Type->T, ParentType-> IDesignModeFacade >
T > FormControl<T>, new()
DesignMode:: FormControl
< Type->ReportDesign, ParentType-> Report >
DesignMode:: ReportDesign
< Type->Method, ParentType-> Class >
DesignMode::Method
«property»
+ Name() :String
+ Source() :String
< T->FormButtonControl >
DesignMode:: FormButtonControl
< T->FormGridControl >
DesignMode:: FormGridControl
Figure 17: ERP DesignTime Mode Facade Class Diagram

ERP DesignTime Mode Façade Construction Sequence Diagram

Every TestSuite has a corresponding DesignModeFacade that it validates. Construction sequence is shown in Figure 18, where in DesignModeFactory determines the type of façade to instantiate based on the façade type that is bound to generic type parameter part of the factory method – ‘Create’. Main difference between a creation of proxy from previous section and façade as explained in this section is that a façade can be created as long as the name of the design time TreeNode is known. In case of a child TreeNode, its parent is inferred from the ‘container’ TestSuite that is provided to Façade factory. The design time TreeNode name is inferred based on the corresponding ERP Metamodel element that was created and cached during model construction phase.

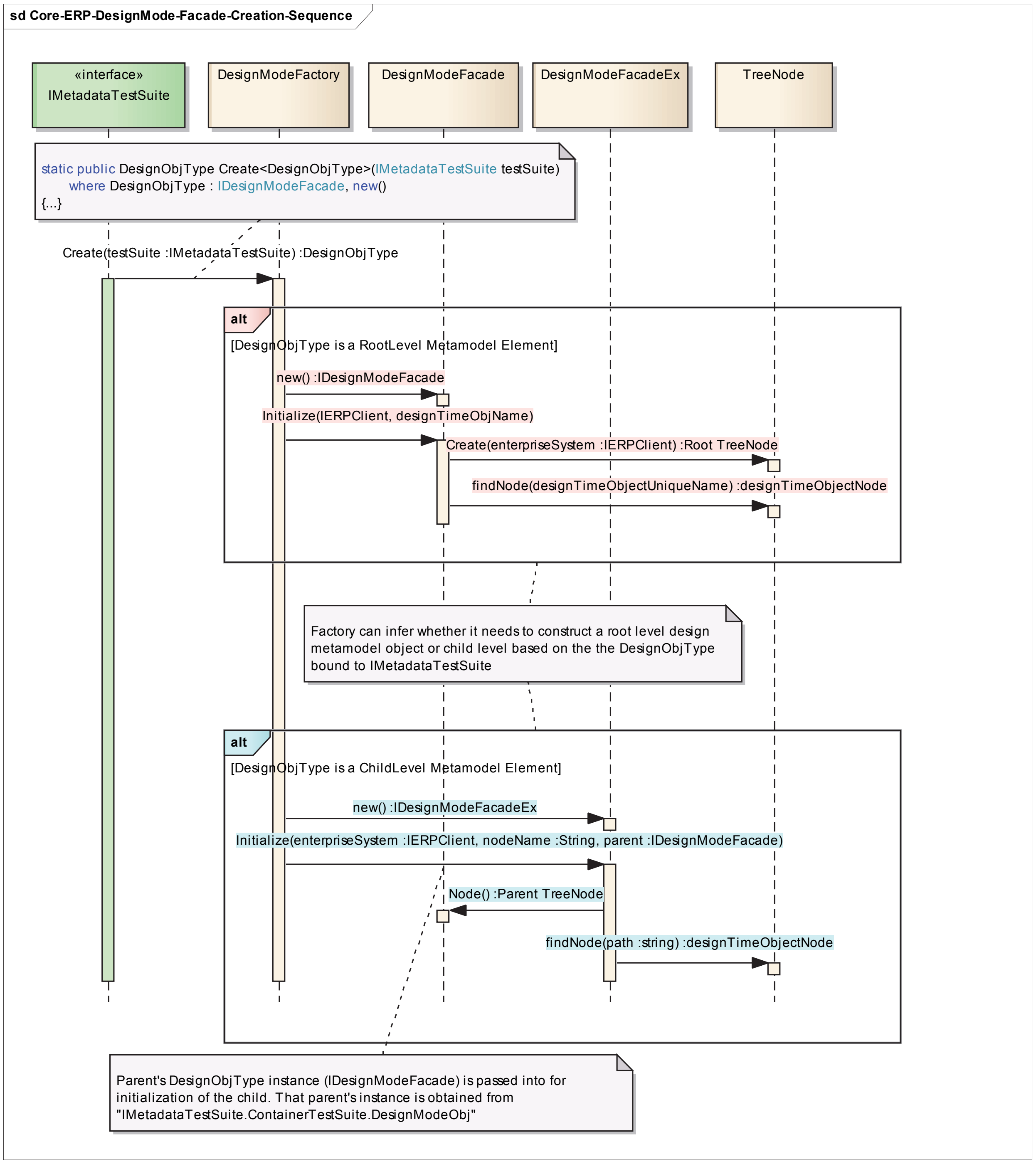

sd Core-ERP-DesignMode-Facade-Creation-Sequence
«interface» IMetadataTestSuite
DesignModeFactory
DesignModeFacade
DesignModeFacadeEx
TreeNode
static public DesignObjType Create<DesignObjType>(IMetadataTestSuite testSuite)
where DesignObjType : IDesignModeFacade, new()
{...}
Create(testSuite :IMetadataTestSuite) :DesignObjType
alt
[DesignObjType is a RootLevel Metamodel Element]
new() :IDesignModeFacade
Initialize(IERPClient, designTimeObjName)
Create(enterpriseSystem :IERPClient) :Root TreeNode
findNode(designTimeObjectUniqueName) :designTimeObjectNode
Factory can infer whether it needs to construct a root level design metamodel object or child level based on the the DesignObjType bound to IMetadataTestSuite
alt
[DesignObjType is a ChildLevel Metamodel Element]
new() :IDesignModeFacadeEx
Initialize(enterpriseSystem :IERPClient, nodeName :String, parent :IDesignModeFacade)
Node() :Parent TreeNode
findNode(path :string) :designTimeObjectNode
Parent's DesignObjType instance (IDesignModeFacade) is passed into for initialization of the child. That parent's instance is obtained from "IMetadataTestSuite.ContainerTestSuite.DesignModeObj"

Figure 18: ERP DesignTime Mode Façade Construction Sequence Diagram

It is not necessary that every ERP Metamodel Element has a corresponding design time node. For example, Label element in one of the ERP products does not have a corresponding design time node. To support such scenario as well as to ensure that DesignModeFacade's collaboration with a TestSuite, *Null Object Pattern* is utilized to design NullDesignModeAdapter which plays the role of a *Null Object* as shown in Figure 19 below.

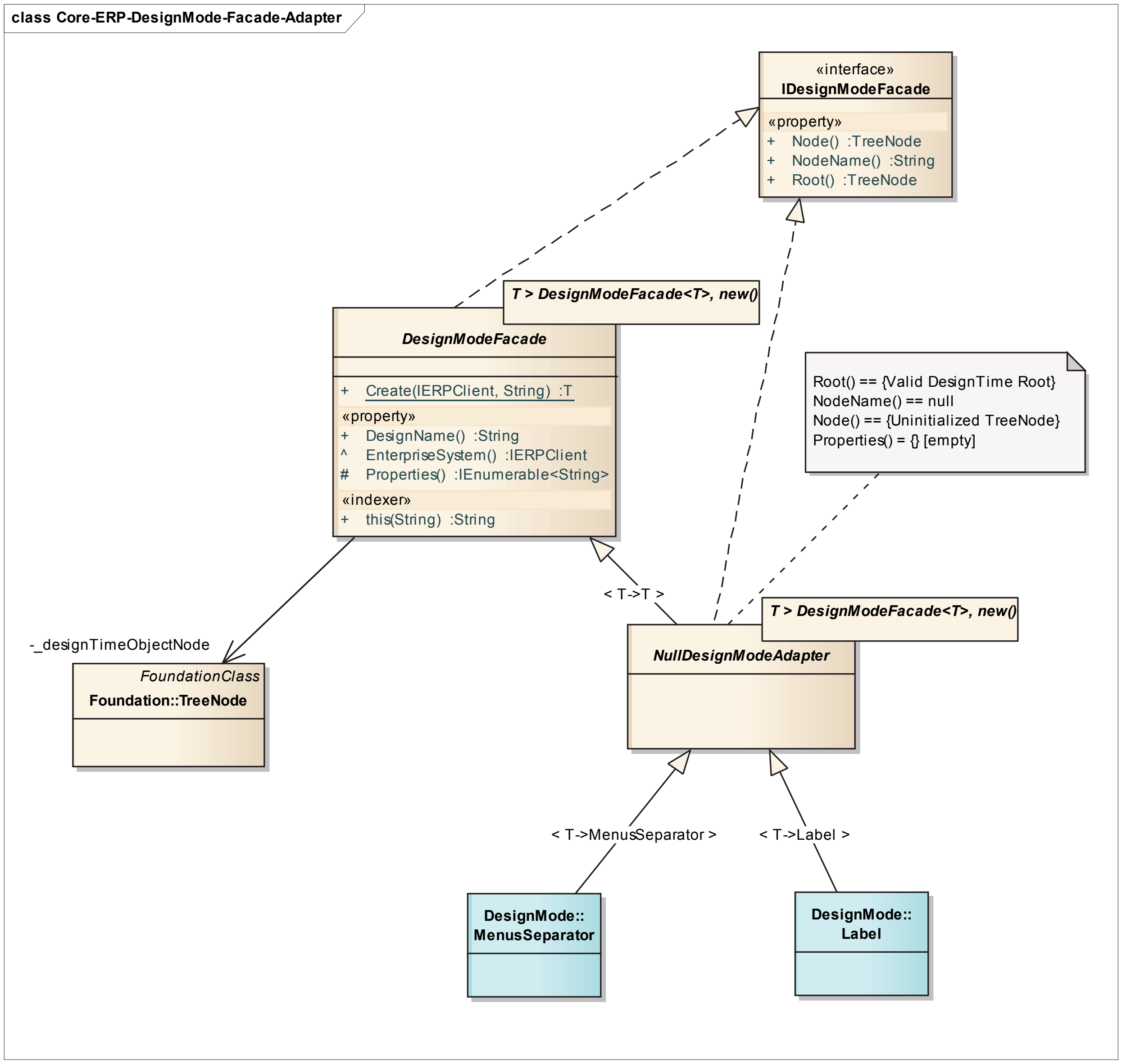


Figure 19: Null Design Mode Facade Adapter

TestServices Package

In earlier sections under Detail Design, various aspects describe how to retrieve information from ERP Client, that is, information accessible via ERP Foundation Classes and

ERP Design Mode Façades. This section focuses on the core functionality supported by EMTF to implement an ERP Metadata Test Suite.

Generally, in a traditional approach such a test is implemented in a procedural manner where in all the steps needed are coded in sequence. However, in order to validate a complex ERP Metamodel, following such an approach, a tester would need to flatten out complex hierarchies; such flattened tests would be laborious to implement and error prone, let alone the poor maintainability. In order to solve that challenge abstractions in this package enable a tester to implement tests in an object oriented manner that are closer to real world model of problem. This real world model is to focus on implementing tests for only one ERP Metamodel Element and move on to next one; and the plumbing that is needed to ensure that the hierarchical dependencies are traversed correctly is now the responsibility of EMTF and not the testers as long as one follows the test pattern described in this paper. Next subsections cover aspects that help implement object oriented tests that are easier to implement and maintain.

ERP Metadata TestSuite Class Diagram

In Figure 20 BaseMetadataTestSuite defines the core responsibilities of an ERP Metadata TestSuite. A TestSuite focuses on testing one ERP Metamodel Element by retrieving its corresponding state, via ERP Foundation Proxy as well as ERP Design Mode Façade, and then validating that information using the cached details that were used to construct the corresponding ERP Metamodel part of a larger ERP Model during the Test Assembly setup phase. Therefore a BaseMetadataTestSuite has three generic type parameters: ElementType, ERPClassType and DesignObjType.

In order to support construction of an ERP Model Element methods part of IModelElementProvider interface must be implemented by a Test Suite. 'CreateModelElement' and ‘CreateModelElementExtensions’ API is used to create an ERP Model Element and its children respectively, like wise ‘AddModelElement’ and ‘AddModelElementExtensions’ is used to hydrate an existing ERP Model Element. In order to construct ERP Model Element extensions a Test Suites creates a composite Test Suite that knows how handle an ERP Model Element of interest. IMetadataTestSuite interface defines CreateExtension and AddExtension that is used by a parent composite TestSuite to build its children or ‘Extensions’ during ERP Model construction. Test Suites hierarchy maps one to one with the hierarchy of the ERP Model Elements, instead of having to flatten out a complex hierarchy into a single monolith test script. An ExtensionTestSuite is cached with its parent TestSuite as and when an extension is created or added. If a TestSuite has a dependency on an ERP Model Element that was already constructed by another TestSuite then the latter is simply added as an Extension. BaseMetadataTestSuite encapsulates necessary logic to create, add and cache TestSuite without derived TestSuites having to worry about the complexities. A TestSuite that corresponds to a root level ERP Model Element that does not have any container is called as RootTestSuite. ExtensionTestSuite is one that has a container TestSuite that either has created or added it as an extension. A RootTestSuite can be an extension of another TestSuite. A parent Test Suite manages the lifetime of its ExtensionTestSuite if it had created it, the relationship here being a ‘composition’ one versus an ‘aggregate’ one in case an existing TestSuite is added as an extension. Figure 12 covered earlier depicts how an ERP Model is constructed that results in creation or addition of a TestSuite as an extension.

The main objective of an ERP Metadata TestSuite at the very least is to ‘Verify MetadataRead’ and ‘VerifyDesignProperties’ (as defined in IERPSystemDesignModeTest contract); this requires a TestSuite to verify both ERP Foundation Class API and ERP Design

Mode Classes. In order to support the verification of state, BaseMetadataTestSuite provides strongly typed access to its corresponding Proxy, Façade as well as cached Model Element state – that is used as expected state – by exposing these APIs respectively: ERPClassObj(), DesignModeObj() and ExpectedObj().

BaseMetadataTestSuite in addition to supporting core ERP Metadata verification requirements also supports integration with any external Test Harness framework for e.g. NUnit. Test Harness is the one that actually drives the tests and reports the test results. In order to ensure tests are resilient to changes in external Test Harness interfaces, BaseMetadataTestSuite introduces a layer of indirection for commonly used test API as exposed by the interfaces ITestEvaluator and ITestLog, available for all derived classes to assert the state of target properties, log exceptions and so on.

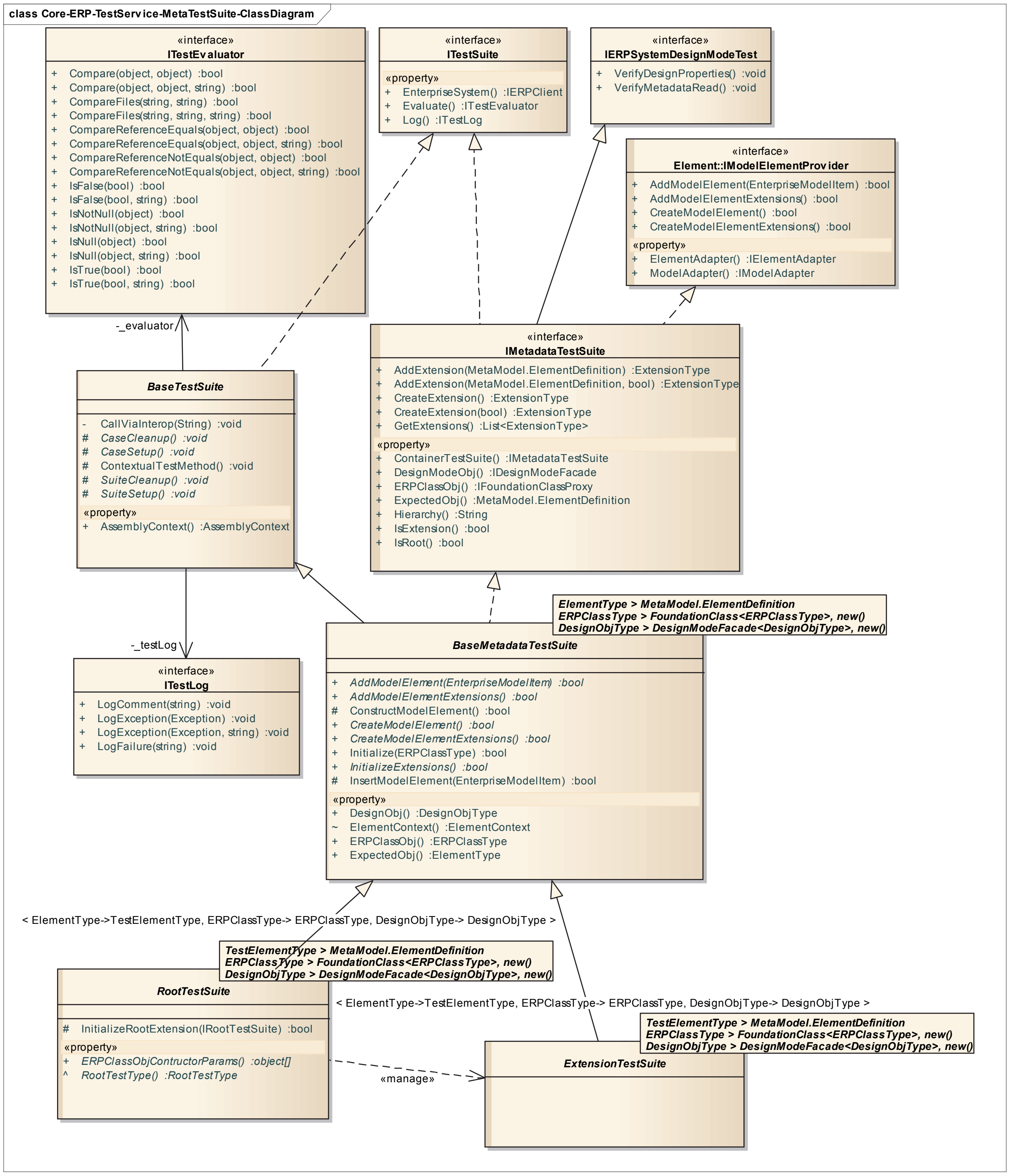

class Core-ERP-TestService-MetaTestSuite-ClassDiagram
«interface» ITestEvaluator
+ Compare(object, object) :bool
+ Compare(object, object, string) :bool
+ CompareFiles(string, string) :bool
+ CompareFiles(string, string, string) :bool
+ CompareReferenceEquals(object, object) :bool
+ CompareReferenceEquals(object, object, string) :bool
+ CompareReferenceNotEquals(object, object) :bool
+ CompareReferenceNotEquals(object, object, string) :bool
+ IsFalse(bool) :bool
+ IsFalse(bool, string) :bool
+ IsNotNull(object) :bool
+ IsNotNull(object, string) :bool
+ IsNull(object) :bool
+ IsNull(object, string) :bool
+ IsTrue(bool) :bool
+ IsTrue(bool, string) :bool
«interface» ITestSuite
«property»
+ EnterpriseSystem() :IERPClient
+ Evaluate() :ITestEvaluator
+ Log() :ITestLog
«interface» IERPSystemDesignModeTest
+ VerifyDesignProperties() :void
+ VerifyMetadataRead() :void
«interface» Element::IModelElementProvider
+ AddModelElement(EnterpriseModelItem) :bool
+ AddModelElementExtensions() :bool
+ CreateModelElement() :bool
+ CreateModelElementExtensions() :bool
«property»
+ ElementAdapter() :IElementAdapter
+ ModelAdapter() :IModelAdapter
-_evaluator
BaseTestSuite
- CallViaInterop(String) :void
# CaseCleanup() :void
# CaseSetup() :void
# ContextualTestMethod() :void
# SuiteCleanup() :void
# SuiteSetup() :void
«property»
+ AssemblyContext() :AssemblyContext
«interface» IMetadataTestSuite
+ AddExtension(MetaModel.ElementDefinition) :ExtensionType
+ AddExtension(MetaModel.ElementDefinition, bool) :ExtensionType
+ CreateExtension() :ExtensionType
+ CreateExtension(bool) :ExtensionType
+ GetExtensions() :List<ExtensionType>
«property»
+ ContainerTestSuite() :IMetadataTestSuite
+ DesignModeObj() :IDesignModeFacade
+ ERPClassObj() :IFoundationClassProxy
+ ExpectedObj() :MetaModel.ElementDefinition
+ Hierarchy() :String
+ IsExtension() :bool
+ IsRoot() :bool
ElementType > MetaModel.ElementDefinition
ERPClassType > FoundationClass<ERPClassType>, new()
DesignObjType > DesignModeFacade<DesignObjType>, new()
-_testLog
«interface» ITestLog
+ LogComment(string) :void
+ LogException(Exception) :void
+ LogException(Exception, string) :void
+ LogFailure(string) :void
BaseMetadataTestSuite
+ AddModelElement(EnterpriseModelItem) :bool
+ AddModelElementExtensions() :bool
# ConstructModelElement() :bool
+ CreateModelElement() :bool
+ CreateModelElementExtensions() :bool
+ Initialize(ERPClassType) :bool
+ InitializeExtensions() :bool
# InsertModelElement(EnterpriseModelItem) :bool
«property»
+ DesignObj() :DesignObjType
~ ElementContext() :ElementContext
+ ERPClassObj() :ERPClassType
+ ExpectedObj() :ElementType
< ElementType->TestElementType, ERPClassType-> ERPClassType, DesignObjType-> DesignObjType >
TestElementType > MetaModel.ElementDefinition
ERPClassType > FoundationClass<ERPClassType>, new()
DesignObjType > DesignModeFacade<DesignObjType>, new()
RootTestSuite
# InitializeRootExtension(IRootTestSuite) :bool
«property»
+ ERPClassObjContructorParams() :object[]
^ RootTestType() :RootTestType
< ElementType->TestElementType, ERPClassType-> ERPClassType, DesignObjType-> DesignObjType >
TestElementType > MetaModel.ElementDefinition
ERPClassType > FoundationClass<ERPClassType>, new()
DesignObjType > DesignModeFacade<DesignObjType>, new()
«manage»
ExtensionTestSuite

Figure 20: ERP Metadata TestSuite Class Diagram

Following sub-sections details key collaborations among abstractions of Figure 20 that happen in each of TestSuite’s phases as mentioned in Figure 5: Setup, Execution, Cleanup.

While Test Harness is responsible starting test automation run, Test Driver is responsible for directing various TestSuite phases for all RootTestSuites as shown in Figure 5. Each RootTestSuite manages the state of its ExtenstionTestSuite in an hierarchical manner. Next subsection covers both TestSuite Setup and CleanUp phase.

ERP Metadata TestSuite Setup Phase Sequence Diagram

Having established the hierarchy of Test Suites during the ERP Model construction, during Test Suite setup phase (Figure 21) the hierarchy of those Test Suites are initialized via depth first traversal as shown in the figure below. Every derived Test Suite must implement ‘InitializeExtensions’ abstract method declared in BaseMetadata Test Suite. Within that method a TestSuite must initialize an ExtensionTestSuite’s ERP Foundation Class Proxy and ERP DesignMode Façade. In case of a RootTestSuite it must also implement ‘ERPClassObjContructorParams()’ that is used by BaseMetadataTestSuite in initializing ERP Foundation Class Proxy and ERP Design Mode Façade for corresponding RootTestSuite. An initialized ExtensionTestSuite is the cached in a deterministic order so that when it comes to Test Suite clean up, each of those Extensions are called in reverse order to execute any overridden SuiteCleanup methods as shown in Figure 22.

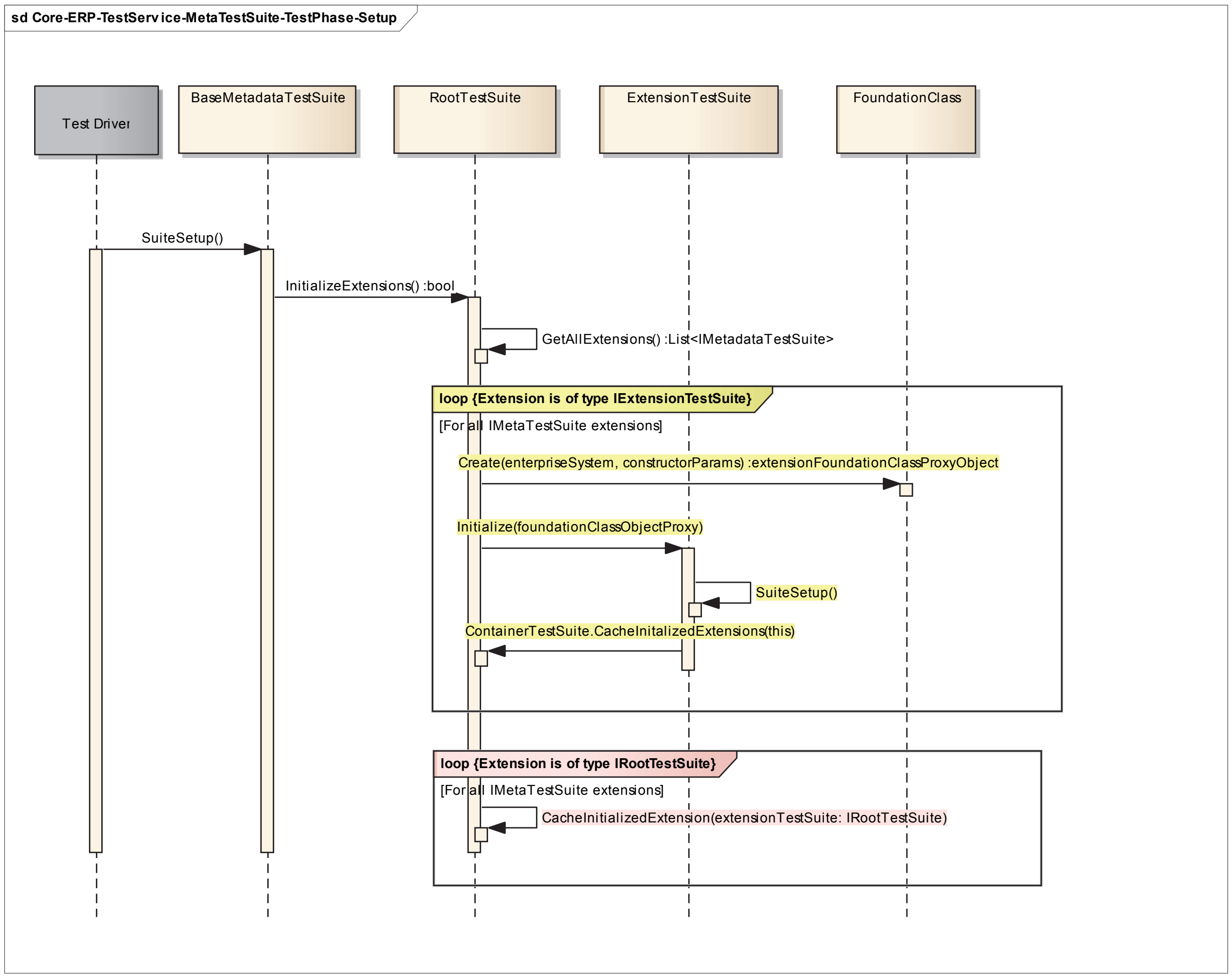

sd Core-ERP-TestService-MetaTestSuite-TestPhase-Setup
Test Driver
BaseMetadataTestSuite
RootTestSuite
ExtensionTestSuite
FoundationClass
SuiteSetup()
InitializeExtensions() :bool
GetAllExtensions() :List<IMetadataTestSuite>
loop {Extension is of type IExtensionTestSuite}
[For all IMetaTestSuite extensions]
Create(enterpriseSystem, constructorParams) :extensionFoundationClassProxyObject
Initialize(foundationClassObjectProxy)
SuiteSetup()
ContainerTestSuite.CacheInitalizedExtensions(this)
loop {Extension is of type IRootTestSuite}
[For all IMetaTestSuite extensions]
CacheInitializedExtension(extensionTestSuite: IRootTestSuite)

Figure 21: ERP Metadata Testsuite Setup Phase Sequence Diagram

ERP Metadata Testsuite Cleanup Phase Sequence Diagram

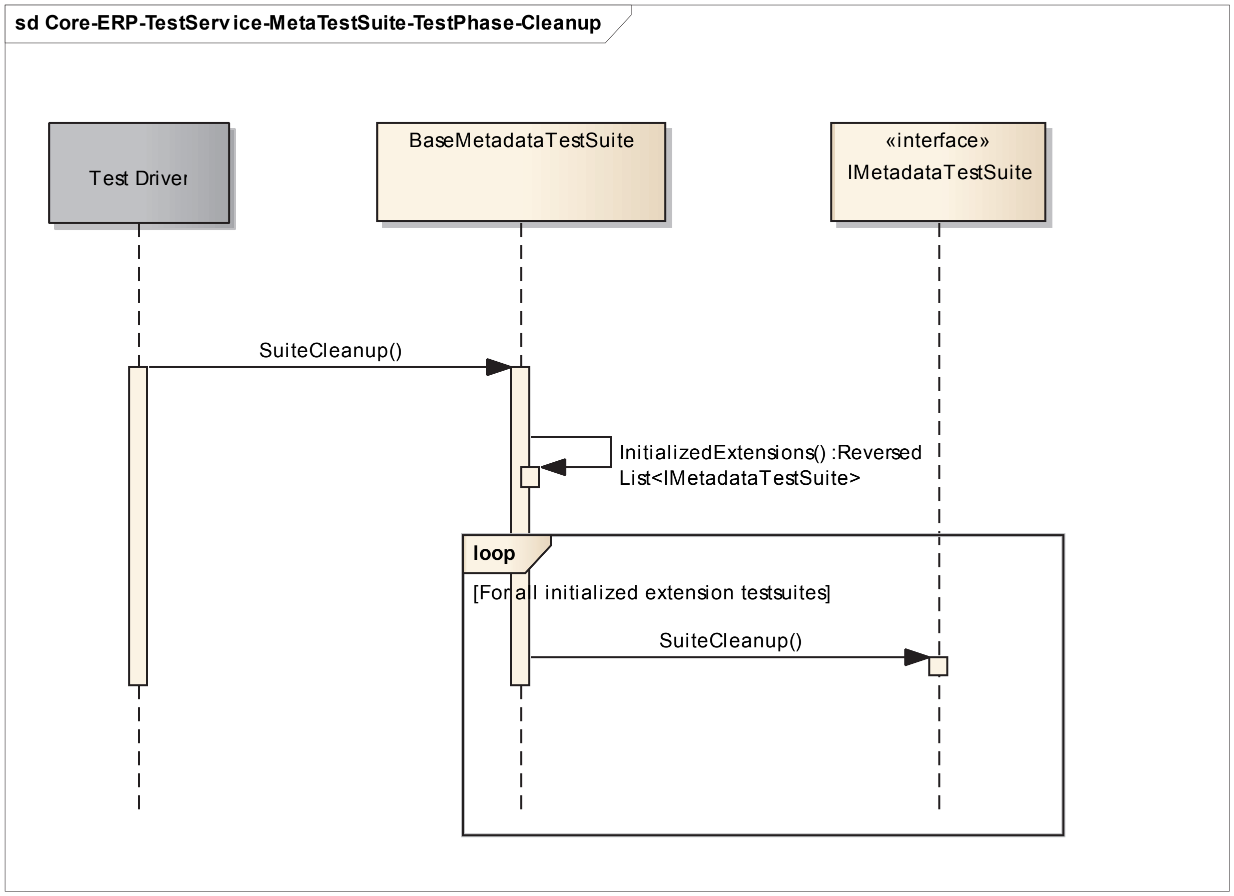

sd Core-ERP-TestServ ice-MetaTestSuite-TestPhase-Cleanup
Test Driver
BaseMetadataTestSuite
«interface»
IMetadataTestSuite
SuiteCleanup()
InitializedExtensions() :Reversed
List<IMetadataTestSuite>
loop
[For all initialized extension testsuites]
SuiteCleanup()

Figure 22: ERP Metadata TestSuite Cleanup Phase Sequence Diagram

At the end of TestSuites Setup phase, ERP Models are deployed by the test framework to ERP Model Repository. Test Driver then starts Execution phase for RootTestSuites in the same order Setup phase was traversed. Next section covers the TestSuite Execution phase in more detail.

ERP Metadata TestSuite Execution Phase Sequence Diagram

Based on the hierarchy of a RootTestSuite that was established during ERP Model construction, CaseSetup is called for all TestSuites in the hierarchy in a top down manner using depth first traversal and only then a verification method on all TestSuites is called the same hierarchical order.

Discovery of test methods within a RootTestSuite is possible only if it implements an interface that declares all the test methods and that interface must be "marked" (using programming language specific 'annotations'); all ExtensionTestSuites are also required to implement that test interface. Due to the fact BaseMetadataTestSuite implements IERPSystemDesignModeTest (Figure 20) interface all the TestSuites must implement those tests at the very least.

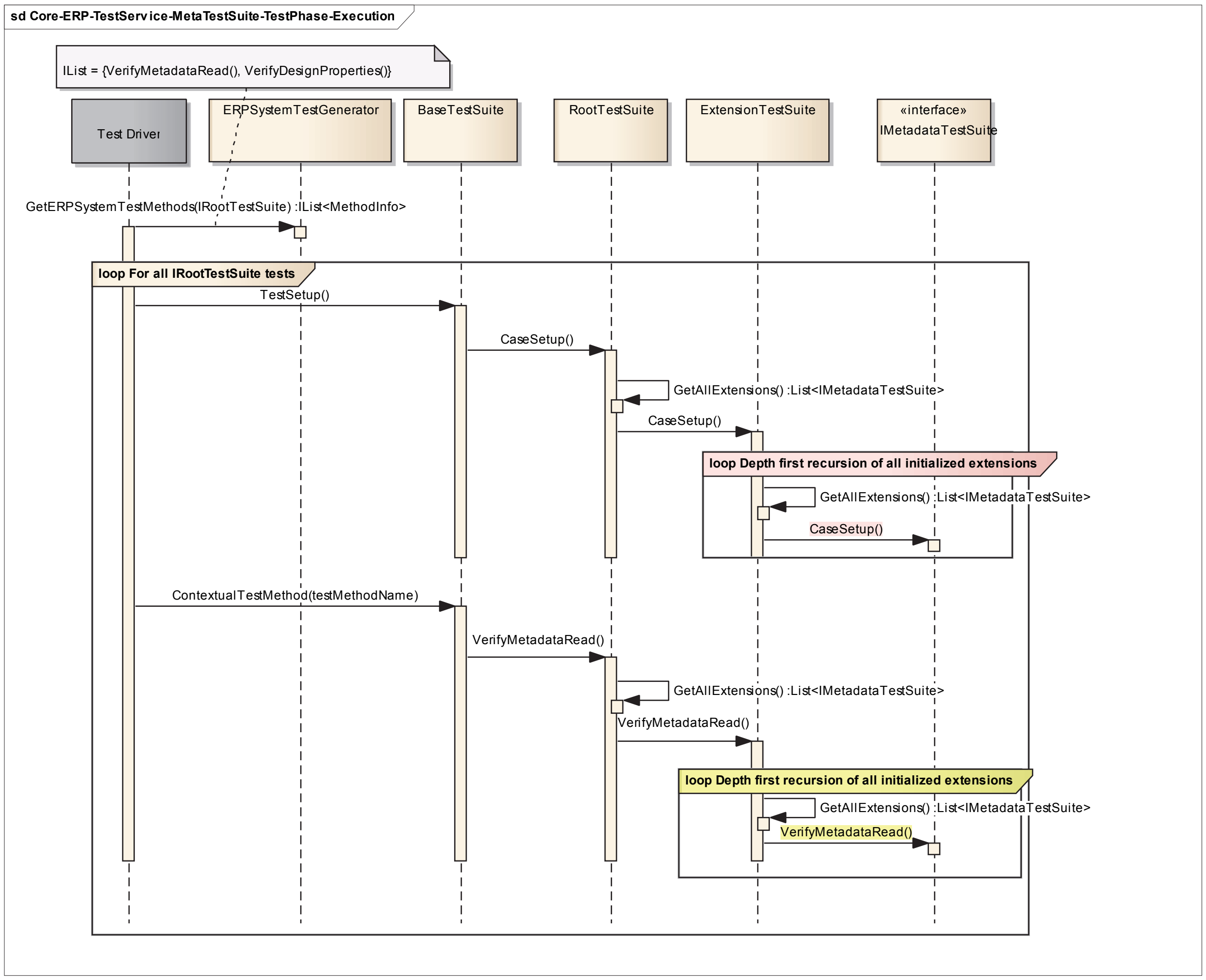

sd Core-ERP-TestService-MetaTestSuite-TestPhase-Execution
IList = {VerifyMetadataRead(), VerifyDesignProperties()}
Test Driver
ERPSystemTestGenerator
BaseTestSuite
RootTestSuite
ExtensionTestSuite
«interface» IMetadataTestSuite
GetERPSystemTestMethods(IRootTestSuite) :IList<MethodInfo>
loop For all IRootTestSuite tests
TestSetup()
CaseSetup()
GetAllExtensions() :List<IMetadataTestSuite>
CaseSetup()
loop Depth first recursion of all initialized extensions
GetAllExtensions() :List<IMetadataTestSuite>
CaseSetup()
ContextualTestMethod(testMethodName)
VerifyMetadataRead()
GetAllExtensions() :List<IMetadataTestSuite>
VerifyMetadataRead()
loop Depth first recursion of all initialized extensions
GetAllExtensions() :List<IMetadataTestSuite>
VerifyMetadataRead()

Figure 23: ERP Metadata Testsuite Execution Phase Sequence Diagram

Thus far the design covers TestSuite aspects that must be implemented by a test automation so that tests can verify metadata accessible via both Foundation Class and UIMode ERP Client.

In a '*1-tier*' deployment topology where in both Test Harness and test automation share process space with an ERP System under test, for e.g. when interacting with Non-UIMode ERP Client, there isn't any need for inter-process communication to happen and therefore caching of expected Model state is not done by the test framework. However, in a '*2-tier*' deployment topology Test Harness does not share process space with an ERP System under test. For example, in the case of UIMode ERP Client to be tested, ERP Models (test input) are constructed and deployed within a TestHarness process space; this process space is not shared with UIMode ERPClient within whose context the tests are run. Hence in such a case expected metadata state needs to be '*marshalled*' or transported to the ERPClient process space and made available when test executes. Next sub-section covers the design necessary for such a transport.

Test Framework Inter-Process Communication Support

When working with System Under Test (SUT) by loading it's components in the same process as that of a Test Driver, the test state management, for e.g. cached expected state, is quite straightforward. However, when a test needs to validate state within another process, the complexity is far greater. Expected state created within a Test Driver process needs to be made available within the context of another remote process, this involves the overhead of inter-process communication as well as serialization and de-serialization of expected test state from Test Driver process into the remote process. A Test Assembly that comprises a collection of Test Suites which in turn comprises a collection of Test Cases must continue to work in case of interprocess communication so that a tester who implements tests does not have to worry about the underlying complexities. In other words all the collaboration sequences covered earlier on setup, execution and cleanup must simply work even when a Test Driver process is different from a process in which Test Suite resides. In order to support such complexities EMTF provides following design that enable Test Suites implementation decoupled from such complexities.

Test Assembly abstraction provided by EMTF assumes the role of an 'InteropTestAssembly' when ERP Client happens to be a UI Portal. The sequence diagram Figure 24 provides collaboration among other abstractions that play a vital role in setup and cleanup of inter-process environment, referred to as 'Interop' in this document. During Test Assembly step up phase, it instantiates InteropHelper which enables InteropDesignTimeConfig and InteropRunTimeConfig that facilitate necessary setup as well as cleanup of Interop environment. The Interop Environment comprises of Test Driver and UI Mode ERP Client processes; the configuration information required to launch them and perform Interop communication is supported by 'InteropRunTimeConfig'. The actual communication includes transfer of expected test state (constructed and deployed test ERP Models state) across Interop boundary; 'InteropDesignTimeConfig' establishes Interop cache of the ERP Models that is accessible from any process.

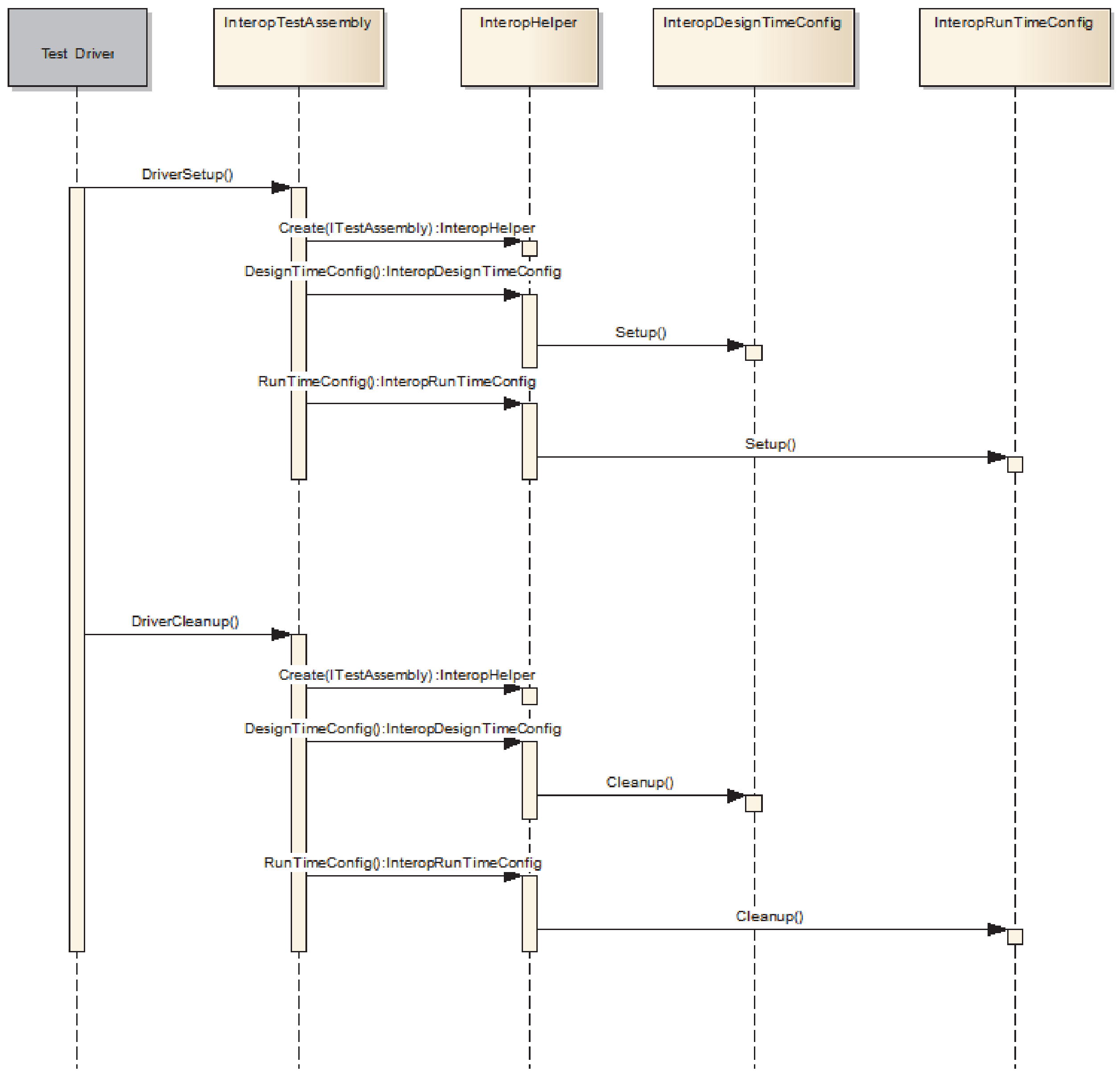

Test Driver
InteropTestAssembly
InteropHelper
InteropDesignTimeConfig
InteropRunTimeConfig
DriverSetup()
Create(ITestAssembly):InteropHelper
DesignTimeConfig():InteropDesignTimeConfig
Setup()
RunTimeConfig():InteropRunTimeConfig
Setup()
DriverCleanup()
Create(ITestAssembly):InteropHelper
DesignTimeConfig():InteropDesignTimeConfig
Cleanup()
RunTimeConfig():InteropRunTimeConfig
Cleanup()

Figure 24: Conceptual Sequence of Interop Interactions

The functionality exposed by InteropDesignTimeConfig that is used by Test Assembly, is designed using *Chain of Responsibility Behavioral Design Pattern*. The 'Handlers' part of that pattern comprise DriverDesignTimeConfig, ERPModelingDesignTimeConfig and UiProcessBridgeDesignTimeConfig, in that succession sequence. These handlers collaborate for DesignTimeConfig setup and cleanup tasks.

DriverDesignTimeConfig helps in management of communication end points between Test Driver and ERP Model Repository. Based on the type of ERP Client it appropriately sets up appropriate channel for communication.

ERPModelingDesignTimeConfig helps in the management of the cache of test ERP Models by setting up a cache that is Interop compatible.

UiProcessBridgeDesignTimeConfig helps in the management of any test state that is needed within the context of a UI Model ERP Client, such as an UI Client Portal. Additionally, it ensures that test ERP Models state is hydrated into expected test state in order to make it ready for tests to access it during verification phase.

Similar to InteropDesignTimeConfig, InteropRunTimeConfig is also designed using *Chain of Responsibility Behavioral Design Pattern*.

DriverRunTimeConfig helps in management of communication end points between Test Driver and ERP Client. Based on the type of ERP Client it appropriately sets up appropriate channel for communication.

ERPModelingRunTimeConfig manages the lifetime of appropriate ERP Client. It include necessary prerequisites needed for ERP Client to be ready to accept communications.

UiProcessBridgeRunTimeConfig helps in the management of various components of ERP System: ERP Client, ERP Server, based on a configuration test network topology.

Testers may extend the functionality of DesignTime and RunTime configuration by implementing InteropTestAssembly; its setup methods are called after all EMTF chain of collaboration (Figure 25) is done. In case of clean up it’s in reverse order, meaning cleanup methods of custom extensions are called before EMTF chain of collaboration.

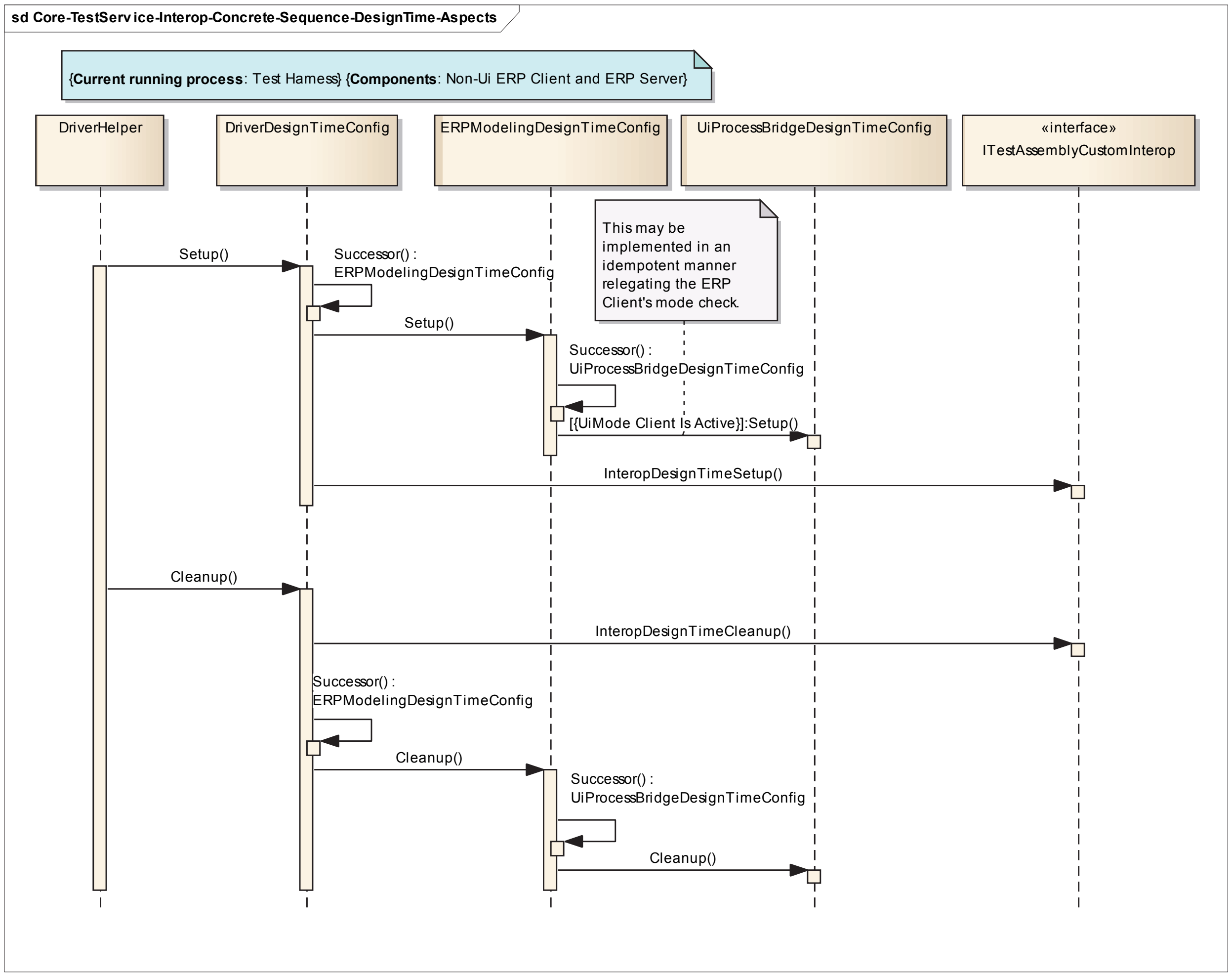

sd Core-TestService-Interop-Concrete-Sequence-DesignTime-Aspects
{Current running process: Test Harness} {Components: Non-Ui ERP Client and ERP Server}
DriverHelper
DriverDesignTimeConfig
ERPModelingDesignTimeConfig
UiProcessBridgeDesignTimeConfig
«interface» ITestAssemblyCustomInterop
This may be implemented in an idempotent manner relegating the ERP Client's mode check.
Setup()
Successor() : ERPModelingDesignTimeConfig
Setup()
Successor() : UiProcessBridgeDesignTimeConfig
[{UiMode Client Is Active}]:Setup()
InteropDesignTimeSetup()
Cleanup()
InteropDesignTimeCleanup()
Successor() : ERPModelingDesignTimeConfig
Cleanup()
Successor() : UiProcessBridgeDesignTimeConfig
Cleanup()

Figure 25: Interop Concrete Sequence DesignTime Aspects

The DesignTime and RunTime Chain of Collaboration among DriverConfig, ERPModeling, UiProcessBridge is established via corresponding InteropHelper chain (DriverHelper, ERPModelingHelper, UiProcessBridgeHelper). Each InteropHelper aggregates both InteropDesignTime and InteropRunTimeConfig, whose successors are defined statically. For example, DriverHelper knows whose its successor helper is and that in turn knows its successor.

Having established necessary prerequisites for Interop communication based on the design described in earlier sections, the next step is passing of test messages across process boundaries (Interop communication). In the figure below ‘local’ region refers to Test Driver process and ‘remote’ refers to UI Mode ERP Client. Due to the length of UML collaboration sequence, it is split into three parts as shown in Figure 26, Figure 27 and Figure 28 respectively. In order to support Interop communications Test Assembly, playing the role of InteropTestAssembly, is loaded not only in Test Driver process memory but also available within UI Mode ERP Client’s process memory.

In Figure 26 (part-1) below, after the launch of UI ERP Client portal by a Test Driver, for each Test Suite, SuiteSetup() method is called. This method needs to execute within the context of the portal, hence BaseMetadataTestSuite recognizes the need to communicate across process collaborates with DriverHelper to transfer the call to the portal process. Before handing off the responsibility to ERPModeling Helper, DriverHelper ensures InteropContext information is current. The InteropContext, carries the details of the method to be called within the remote process. ERPModelingHelper then serializes the test ERP Model state and store it in the InteropContext, constructed within Test Driver process, to be made available in the remote portal process. The call, carrying InteropContext, then crosses process boundary to the portal ERP Client side.

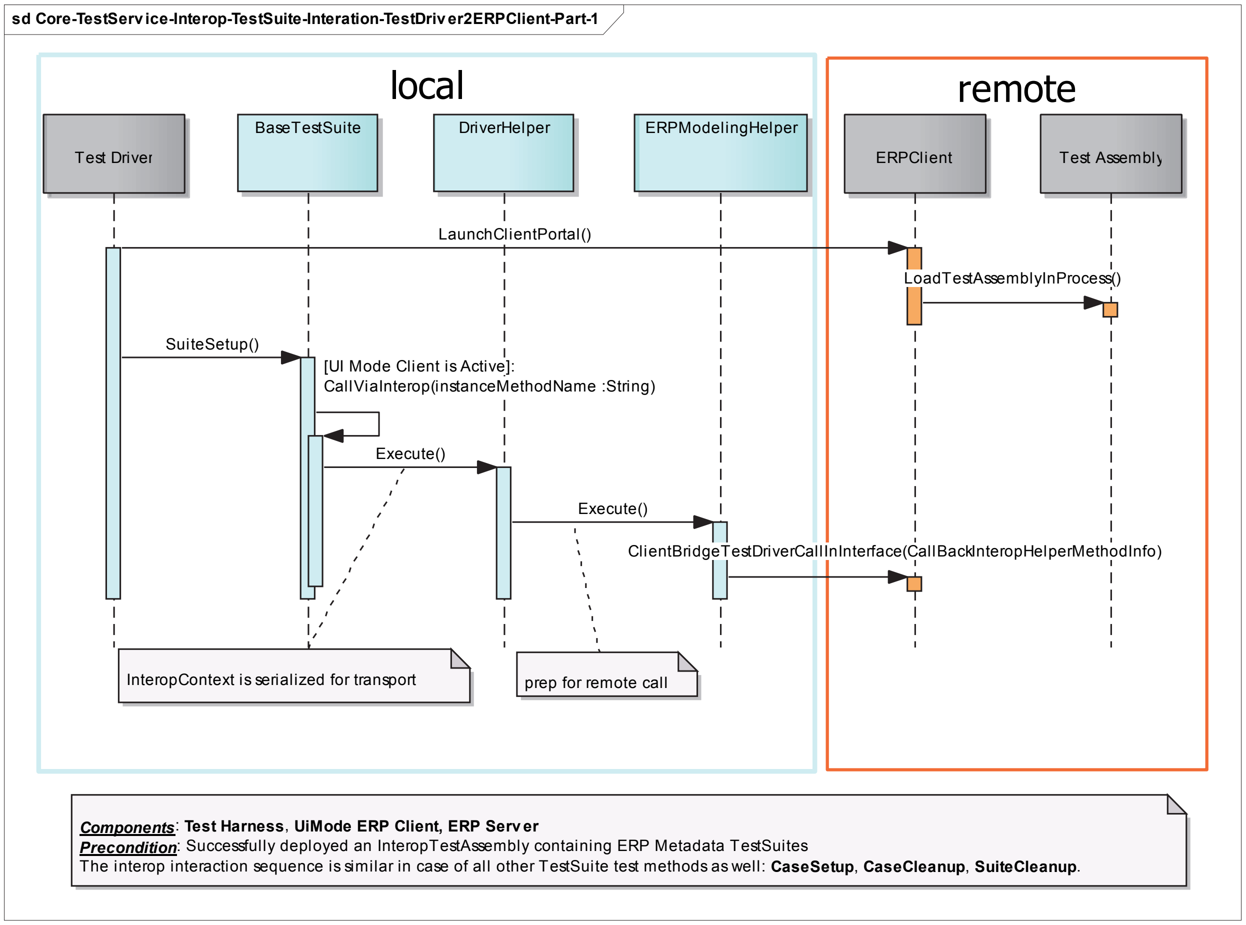

sd Core-TestService-Interop-TestSuite-Interation-TestDriver2ERPClient-Part-1
local
remote
Test Driver
BaseTestSuite
DriverHelper
ERPModelingHelper
ERPClient
Test Assembly
LaunchClientPortal()
LoadTestAssemblyInProcess()
SuiteSetup()
[UI Mode Client is Active]:
CallViaInterop(instanceMethodName :String)
Execute()
Execute()
ClientBridgeTestDriverCallInInterface(CallBackInteropHelperMethodInfo)
InteropContext is serialized for transport
prep for remote call
Components: Test Harness, UiMode ERP Client, ERP Server
Precondition: Successfully deployed an InteropTestAssembly containing ERP Metadata TestSuites
The interop interaction sequence is similar in case of all other TestSuite test methods as well: CaseSetup, CaseCleanup, SuiteCleanup.

Figure 26: TestDriver to ERPClient Call across Process Boundary to Execute Tests in ERPClient Process Space – Part 1

Portal client on receiving the call (Figure 27, part-2), passes the received message (InteropContext) to InteropTestAssembly. It then collaborates with its corresponding UiProcessBridgeHelper to handle the incoming request. UiProcessBridgeHelper ensures expected test state (ERP Models) is hydrated and then passes the test method specific invocation responsibility to its TestSuiteManager.

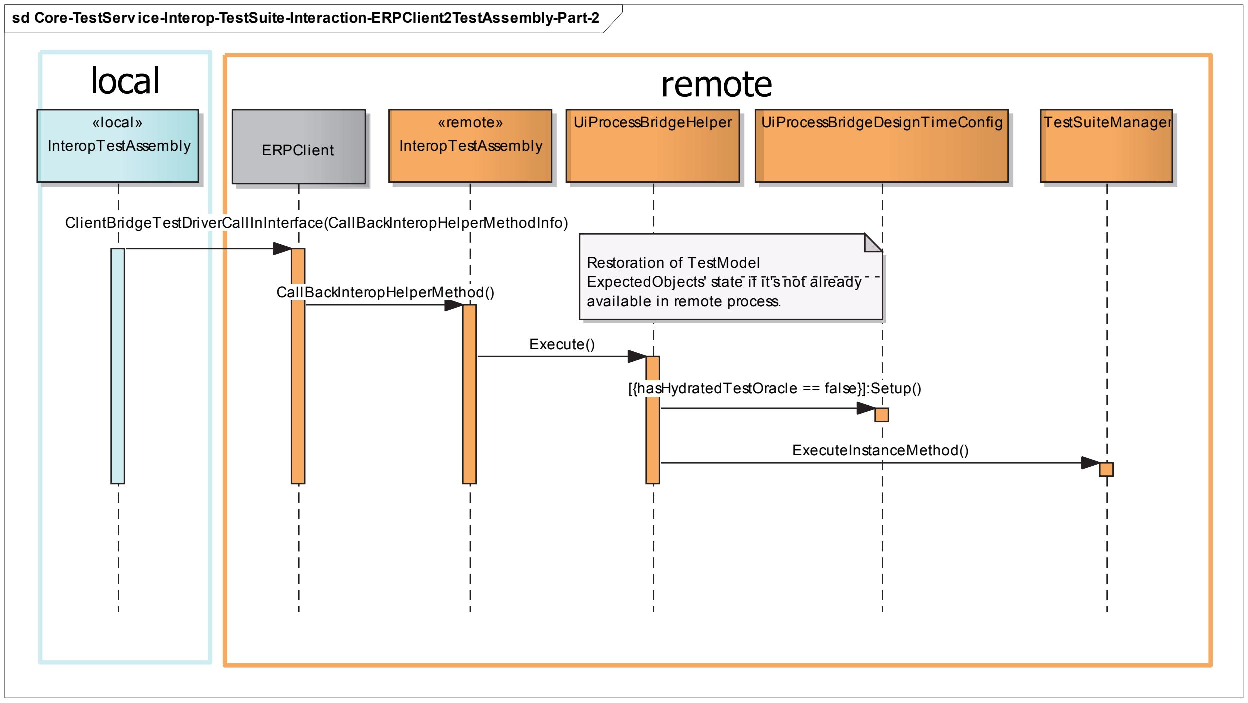

sd Core-TestService-Interop-TestSuite-Interaction-ERPClient2TestAssembly-Part-2
local
remote
«local» InteropTestAssembly
ERPClient
«remote» InteropTestAssembly
UiProcessBridgeHelper
UiProcessBridgeDesignTimeConfig
TestSuiteManager
ClientBridgeTestDriverCallInInterface(CallBackInteropHelperMethodInfo)
CallBackInteropHelperMethod()
Restoration of TestModel ExpectedObjects' state if it's not already available in remote process.
Execute()
[{hasHydratedTestOracle == false}]:Setup()
ExecuteInstanceMethod()

Figure 27: TestDriver to ERPClient Call across Process Boundary to Execute Tests in ERPClient Process Space – Part 2

In Figure 28 (part-3) below, Test Suite Manager on recognizing that it is a remote call retrieves the method to be invoked from InteropContext and then calls that method on specified RootTestSuite. As a result of composite pattern, the method with the same name present in the hierarchy are recursively called starting from RootTestSuite until leaf Test Suite is touched. This Interop collaboration sequence is followed for all messages that Test Driver sends to a Metadata TestSuite; including SuiteSetup, CaseSetup, CaseCleanup, SuiteCleanup, as well as all declared test methods such as VerifyMetadataRead, VerifyDesignProperties.

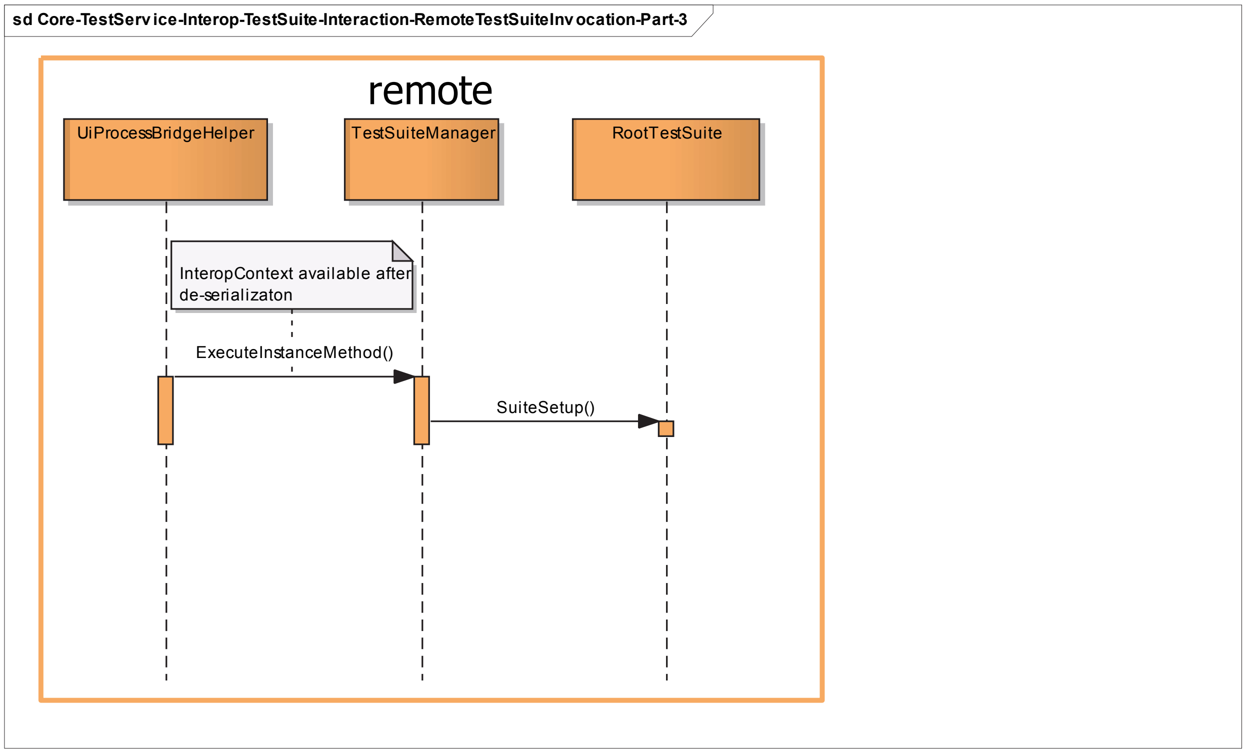

sd Core-TestServ ice-Interop-TestSuite-Interaction-RemoteTestSuiteInv ocation-Part-3
remote
UiProcessBridgeHelper
TestSuiteManager
RootTestSuite
InteropContext available after de-serializaton
ExecuteInstanceMethod()
SuiteSetup()

Figure 28: TestDriver to ERPClient Call across Process Boundary to Execute Tests in ERPClient Process Space – Part 3

ERP METAMODEL TEST SUITE PATTERN

"*Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it in the same way twice.*"

- Christopher Alexander

This chapter distills the information provided in Detail Design and presents the core ERP Metamodel Test Suite Pattern as shown in Figure 29.

BaseMetadataTestSuite's main objective is to test ERP Metadata corresponding to an ERP Metamodel Element. It streamlines access to ERP Metadata via ERPClient by requiring testers to specify ERP Foundation Class Proxy and ERP Design Mode Façade part of a derived Test Suite definition. It also manages the expected test state seamlessly so that testers can focus on essential aspects of an ERP Metadata Test Suite implementation. Additionally, besides transparently providing core functionality for Inter-process execution of test methods, using *Template Method Behavioral Design Pattern* it also defines a scaffolding which covers invariant steps of the algorithms (caching and hydration of ERP Model Elements as well as TestSuite Extensions) and then delegates the variant steps to derived classes, in some cases providing default implementation such as SuiteSetup, CaseSetup, CaseCleanup, SuiteCleanup.

Abstract classes RootTestSuite and ExtensionTestSuite are specializations of Abstract BaseMetadataTestSuite that facilitate building complex hierarchy of TestSuites by either creation of new ones or reuse of existing ones, in a very straightforward by following the principle of divide and conquer. In Figure 29 below all the classes in turquoise are part of EMTF and ones in blue, as an example, is all what a tester needs to implement.

In addition to the core Test Suite pattern, Proxy and Façade patterns (red colored class in Figure 29) were also described as to how information can be retrieved from API based ERP Foundation Classes as well as ERP Design Mode Classes. Class representing expected metadata state (cached test oracle) is colored in green.

Although this document doesn’t cover verification of ERP Metadata write functionality, it can be extended for write along the same lines as that of ERP Metadata read.

This design pattern enforces single responsibility, consistency, loose coupling, flexible, and reusable; key characteristics of maintainable software that is resilient to underlying changes in either an ERP Client or a Test Harness framework used to drive the tests.

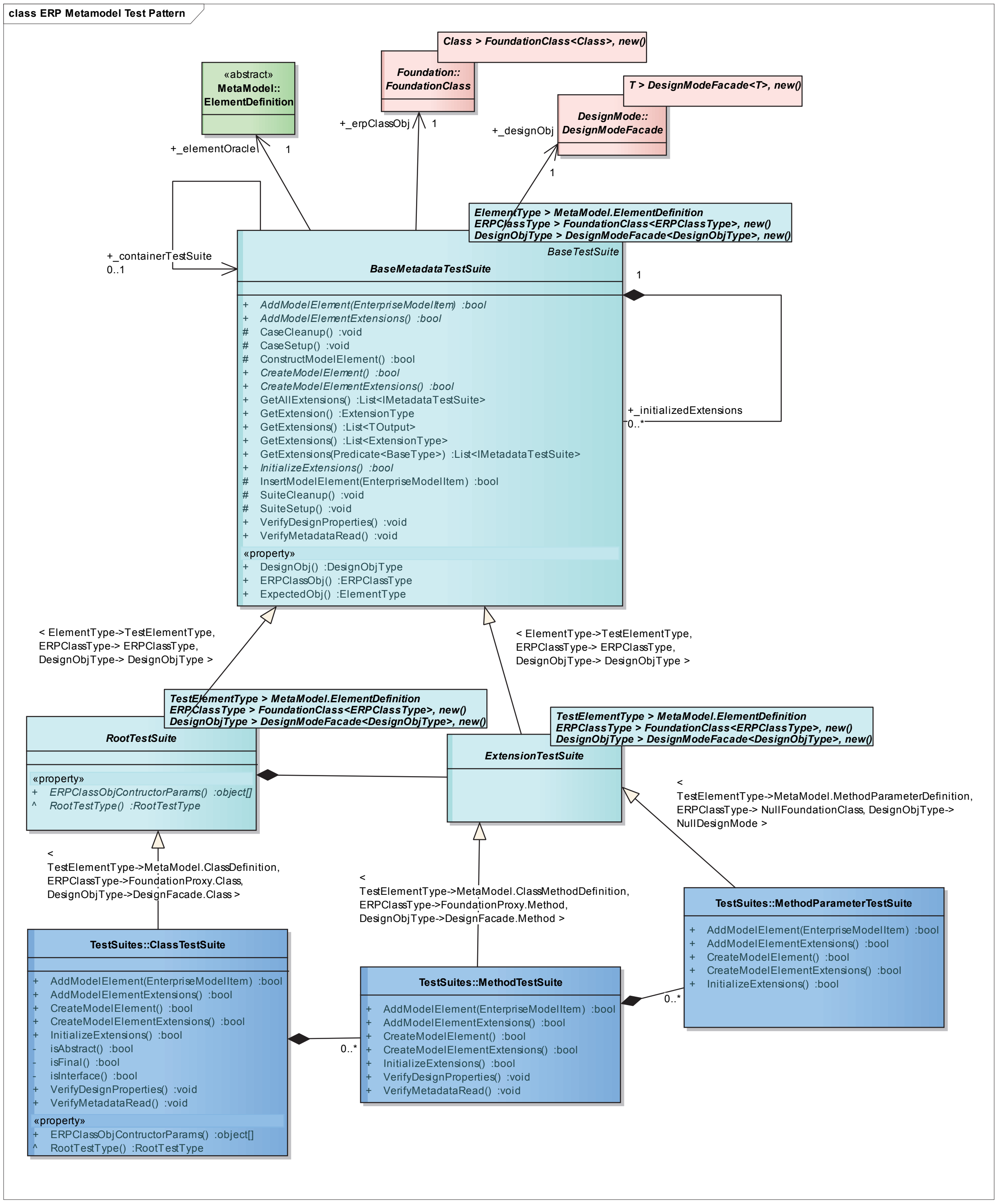

class ERP Metamodel Test Pattern
«abstract» MetaModel::ElementDefinition
Foundation::FoundationClass
Class > FoundationClass<Class>, new()
DesignMode::DesignModeFacade
T > DesignModeFacade<T>, new()
+_elementOracle 1
+_erpClassObj 1
+_designObj 1
+_containerTestSuite 0..1
ElementType > MetaModel.ElementDefinition
ERPClassType > FoundationClass<ERPClassType>, new()
DesignObjType > DesignModeFacade<DesignObjType>, new()
BaseTestSuite
BaseMetadataTestSuite
+ AddModelElement(EnterpriseModelItem) :bool
+ AddModelElementExtensions() :bool
# CaseCleanup() :void
# CaseSetup() :void
# ConstructModelElement() :bool
+ CreateModelElement() :bool
+ CreateModelElementExtensions() :bool
+ GetAllExtensions() :List<IMetadataTestSuite>
+ GetExtension() :ExtensionType
+ GetExtensions() :List<TOutput>
+ GetExtensions() :List<ExtensionType>
+ GetExtensions(Predicate<BaseType>) :List<IMetadataTestSuite>
+ InitializeExtensions() :bool
# InsertModelElement(EnterpriseModelItem) :bool
# SuiteCleanup() :void
# SuiteSetup() :void
+ VerifyDesignProperties() :void
+ VerifyMetadataRead() :void
«property»
+ DesignObj() :DesignObjType
+ ERPClassObj() :ERPClassType
+ ExpectedObj() :ElementType
1
+_initializedExtensions 0..*
< ElementType->TestElementType, ERPClassType-> ERPClassType, DesignObjType-> DesignObjType >
< ElementType->TestElementType, ERPClassType-> ERPClassType, DesignObjType-> DesignObjType >
TestElementType > MetaModel.ElementDefinition
ERPClassType > FoundationClass<ERPClassType>, new()
DesignObjType > DesignModeFacade<DesignObjType>, new()
RootTestSuite
«property»
+ ERPClassObjContructorParams() :object[]
^ RootTestType() :RootTestType
TestElementType > MetaModel.ElementDefinition
ERPClassType > FoundationClass<ERPClassType>, new()
DesignObjType > DesignModeFacade<DesignObjType>, new()
ExtensionTestSuite
< TestElementType->MetaModel.MethodParameterDefinition, ERPClassType-> NullFoundationClass, DesignObjType-> NullDesignMode >
< TestElementType->MetaModel.ClassDefinition, ERPClassType->FoundationProxy.Class, DesignObjType->DesignFacade.Class >
< TestElementType->MetaModel.ClassMethodDefinition, ERPClassType->FoundationProxy.Method, DesignObjType->DesignFacade.Method >
TestSuites::MethodParameterTestSuite
+ AddModelElement(EnterpriseModelItem) :bool
+ AddModelElementExtensions() :bool
+ CreateModelElement() :bool
+ CreateModelElementExtensions() :bool
+ InitializeExtensions() :bool
TestSuites::ClassTestSuite
+ AddModelElement(EnterpriseModelItem) :bool
+ AddModelElementExtensions() :bool
+ CreateModelElement() :bool
+ CreateModelElementExtensions() :bool
+ InitializeExtensions() :bool
- isAbstract() :bool
- isFinal() :bool
- isInterface() :bool
+ VerifyDesignProperties() :void
+ VerifyMetadataRead() :void
«property»
+ ERPClassObjContructorParams() :object[]
^ RootTestType() :RootTestType
TestSuites::MethodTestSuite
+ AddModelElement(EnterpriseModelItem) :bool
+ AddModelElementExtensions() :bool
+ CreateModelElement() :bool
+ CreateModelElementExtensions() :bool
+ InitializeExtensions() :bool
+ VerifyDesignProperties() :void
+ VerifyMetadataRead() :void
0..*
0..*

Figure 29: ERP Metamodel TestSuite Pattern

UNIT TEST DESIGN

"*Quality is never an accident; it is always the result of high intention, sincere effort, intelligent direction and skillful execution; it represents the wise choice of many alternatives.*"

– William A. Foster

In order to test the collaboration among core abstractions part of ERP Metadata TestSuite Pattern (Figure 29) the unit test design pattern shown in Figure 30 was followed. Following core unit test scenarios were validated:

1. Standalone RootTestSuite without any ExtensionTestSuite
2. A RootTestSuite which composes an ExtensionTestSuite
3. A RootTestSuite that composes an ExtensionTestSuite which in turn composes an ExtenstionTestSuite
4. A RootTestSuite which aggregates another RootTestSuite
5. A RootTestSuite that aggregates another RootTestSuite which in turn composes another ExtenstionTestSuite
6. A RootTestSuite aggregrates top level RootTestSuites from previous tests {1, 2, 3, 4, 5}.

All the Test Suites implemented part of unit tests are decoupled from ERP Metadata API via usage of *Null Object Pattern*.

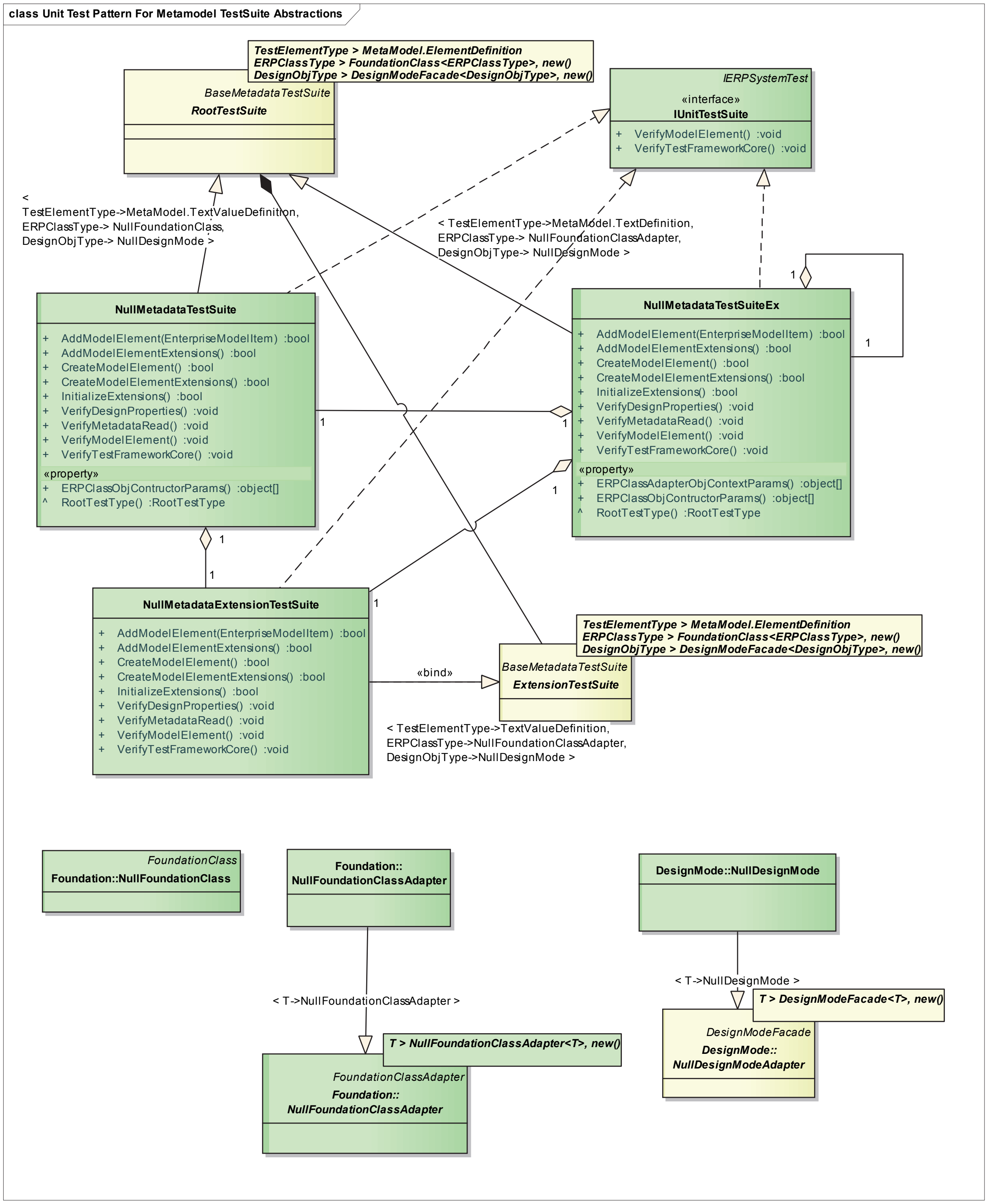

class Unit Test Pattern For Metamodel TestSuite Abstractions
TestElementType > MetaModel.ElementDefinition
ERPClassType > FoundationClass<ERPClassType>, new()
DesignObjType > DesignModeFacade<DesignObjType>, new()
BaseMetadataTestSuite
RootTestSuite
IERPSystemTest
«interface»
IUnitTestSuite
+ VerifyModelElement() :void
+ VerifyTestFrameworkCore() :void
< TestElementType->MetaModel.TextValueDefinition, ERPClassType-> NullFoundationClass, DesignObjType-> NullDesignMode >
< TestElementType->MetaModel.TextDefinition, ERPClassType-> NullFoundationClassAdapter, DesignObjType-> NullDesignMode >
NullMetadataTestSuite
+ AddModelElement(EnterpriseModelItem) :bool
+ AddModelElementExtensions() :bool
+ CreateModelElement() :bool
+ CreateModelElementExtensions() :bool
+ InitializeExtensions() :bool
+ VerifyDesignProperties() :void
+ VerifyMetadataRead() :void
+ VerifyModelElement() :void
+ VerifyTestFrameworkCore() :void
«property»
+ ERPClassObjContructorParams() :object[]
^ RootTestType() :RootTestType
NullMetadataTestSuiteEx
+ AddModelElement(EnterpriseModelItem) :bool
+ AddModelElementExtensions() :bool
+ CreateModelElement() :bool
+ CreateModelElementExtensions() :bool
+ InitializeExtensions() :bool
+ VerifyDesignProperties() :void
+ VerifyMetadataRead() :void
+ VerifyModelElement() :void
+ VerifyTestFrameworkCore() :void
«property»
+ ERPClassAdapterObjContextParams() :object[]
+ ERPClassObjContructorParams() :object[]
^ RootTestType() :RootTestType
NullMetadataExtensionTestSuite
+ AddModelElement(EnterpriseModelItem) :bool
+ AddModelElementExtensions() :bool
+ CreateModelElement() :bool
+ CreateModelElementExtensions() :bool
+ InitializeExtensions() :bool
+ VerifyDesignProperties() :void
+ VerifyMetadataRead() :void
+ VerifyModelElement() :void
+ VerifyTestFrameworkCore() :void
«bind»
BaseMetadataTestSuite
ExtensionTestSuite
< TestElementType->TextValueDefinition, ERPClassType->NullFoundationClassAdapter, DesignObjType->NullDesignMode >
1
1
1
1
1
1
1
1
1
FoundationClass
Foundation::NullFoundationClass
Foundation::
NullFoundationClassAdapter
DesignMode::NullDesignMode
< T->NullFoundationClassAdapter >
< T->NullDesignMode >
T > NullFoundationClassAdapter<T>, new()
T > DesignModeFacade<T>, new()
FoundationClassAdapter
Foundation::
NullFoundationClassAdapter
DesignModeFacade
DesignMode::
NullDesignModeAdapter

Figure 30: UnitTest Abstractions to Verify Core TestSuite Pattern

CONCLUSION

Traditional approach to testing core metamodel functionality that is part of a large and complex ERP system has several challenges. An impact of that is poor test coverage due to lengthy and complex test scripts as a result of flattening ERP Model hierarchy; these steps are neither extendible to build related complex test scenarios nor are reusable across different deployment topologies. As traditional test scripts also directly handle complex interactions with distributed ERP system components they tend to be less reliable, affecting adversely regression testing outcome.

In order to alleviate ERP System's metamodel testing challenges, this paper presented an ERP Metamodel Test Framework Pattern. The pattern introduces three core sub-patterns: ERP Model Adapters to manage test inputs as well test oracle, ERP Metamodel API Proxies and Façades to transparently access ERP Metadata. This pattern enables verification of ERP Metamodel functionality in both UI and non-UI modes, across different deployment topologies, using single set of tests. Additionally, it allows test engineers to easily extend intermediate test steps by either specializing existing tests or aggregating existing ones in a strongly typed manner and platform independent way; thereby increasing productivity over traditional approach. And lastly, but most importantly the test automations built using this pattern are resilient to changes in the interface of an ERP System as long as the functionality being tested is still supported. Due to the advantages of this pattern over traditional approach, ERP Metamodel Test Pattern also has positive impact on triple constraints of an ERP Metamodel test project: scope, time and cost, as well.

CONSTRAINTS AND FUTURE WORK

The pattern presented here assumes that like ERP Foundation Classes there will be an API available to access ERP Metadata for UI Client, i.e., there will be ERP Design Mode Class for every ERP Model Element in UI Mode. However, this is not always the case requiring UI Mode metadata to be accessed by simulating human interaction with the ERP Client. While this increases the complexity of EMTF, the tests implemented using this pattern are not impacted if one decided to extend EMTF to support such simulation.

Future enhancement of the pattern presented here would be to extend the design to support simulation of human interaction with UI ERP Client and be able to access UI Mode metadata for elements that do not have API. Conceptually, ERP Design Mode Façade would still be provided to testers but instead of accessing metadata via API, corresponding metadata will be accessed using external human interaction simulation frameworks such as Selenium or any GUI testing framework [28]; and retrieve metadata properties from designer surface. *Strategy Behavioral Design Pattern* can be utilized to either use API or simulate human interaction in order to retrieve ERP Metadata for a metamodel element. Another enhancement would be to enable an option to automatically generate test inputs based on combinatorial testing techniques [29]; and automatically build test ERP Models based on those generated test inputs. Even when EMTF is enhanced in future, existing test automations built using the pattern discussed here would continue to work without needing any change.

REFERENCES

1. E. Gamma, R. Helm, R. Johnson, and J. M. Vlissides. "*Design Patterns. Elements of Reusable Object-Oriented Software*". Addison Wesley, 1994.
2. M. Fowler et al., "*Refactoring: Improving the Design of Existing Code*", Addison-Wesley, 1999.
3. R.V. Binder*, "Testing Object-Oriented Systems",* Addison-Wesley, 2000.
4. A. Abran, P. Bourque, R. Dupuis, J. W. Moore, L. Tripp. *"Guide to the Software Engineering Body of Knowledge",* IEEE Computer Society, 2004.
5. R.S. Pressman, "*Software Engineering: A Practitioner's Approach"*, sixth edition, McGraw-Hill, 2004.
6. S. McConnell, "*Code Complete: A Practical Handbook of Software Construction*", second edition, Microsoft Press, 2004.
7. J.P. Cavano, J.A. McCall, "*A framework for the measurement of software quality*", ACM SIGSOFT Software Engineering Notes, Volume 3 Issue 5, November 1978.
8. G. J. Myers, C. Sandler, and T. Badgett, "*The art of software testing*", John Wiley & Sons, 2012.
9. K. Naik, P. Tripathy, "*Software Testing and Quality Assurance: Theory and Practice*". John Wiley & Sons, 2008.
10. W.E. Perry, "*Effective Methods for Software Testing*", 3rd Edition. Wiley Publishing, 2006.

11. H. Lackner, J. Svacina, S. Weißleder, M. Aigner, and M. Kresse, "*Introducing Model-Based Testing in Industrial Context - An Experience Report*", Workshop on Model-Based Testing in Practice, 2010.
12. P. E. Ammann and J. Offutt, "*Introduction to Software Testing*", Cambridge University Press, 2008.
13. G. Booch, J. Rumbaugh, I. Jacobson, "*The Unified Modeling Language User Guide*", Addison-Wesley, 2001.
14. J. Hartmann, C. Imoberdorf, M. Meisinger, "*UML-Based integration testing*", Proceedings of the 2000 ACM SIGSOFT International Symposium on Software testing and Analysis, 2000.
15. J. Arlow, I. Neustadt, "*Enterprise Patterns and MDA: Building Better Software with Archetype Patterns and UML*", Addison-Wesley, 2004.
16. B. Y., Alkazemi, et al. "On Evaluating the Architecture of ERP Systems", The ResearchBulletin of Jordan ACM, Volume II.
17. T. Barker, M. N. Frolick, "*ERP implementation failure: A case study*", Information Systems Management, *20*(4), 43-49, 2003.
18. J.I. Chen, "*Planning for ERP systems: analysis and future trend*", Business Process Management Journal, Vol. 7, No. 5, 374-386, 2001.
19. M. A. T. Alsudairi. "*Analysis and Exploration of Critical Success Factors of ERP Implementation: A Brief Review*". International Journal of Computer Applications, 44-52, May 2013.

20. J. Rivera, "*Gartner Says By 2016, the Impact of Cloud and Emergence of Postmodern ERP Will Relegate Highly Customized ERP Systems to Legacy Status"*. Gartner, 29 Jan. 2014. Web. 28 Apr. 2014. <http://www.gartner.com/newsroom/id/2658415>.

21. J. Mylopoulos, "*What Is Metamodeling?*", Department of Computer Science, University of Toronto., 2004. Web. 27 Dec. 2012. <http://www.cs.toronto.edu/~jm/2507S/Notes04/Meta.pdf>.

22. "*JPA Metamodel API (Type, ManagedType, EntityType, Attribute)*", ObjectDB Software, Web. 12 Jan. 2013. <http://www.objectdb.com/java/jpa/persistence/metamodel>.

23. "*Using the Metamodel API to Model Entity Classes - The Java EE 6 Tutorial*" Oracle, Web. 12 Jan. 2013. <http://docs.oracle.com/javaee/6/tutorial/doc/gjiup.html>.

24. "*Meta Model of Data Types - SAP Global Data Types [read-only] - SCN Wiki*", SAP, Web. 10 Jan. 2013. <http://wiki.scn.sap.com/wiki/display/GDT/Meta+Model+of+Data+Types>.

25. T. Kühne, "*Matters of (Meta-) Modeling*", Springer-Verlag, Journal on Software and Systems Modeling, Volume 5, 369-385, 2006.

26. W. Hesse, "*More matters on (meta-)modelling: remarks on Thomas Kühne's 'matters'*", Springer-Verlag, Journal on Software and Systems Modeling, Volume 5, 387-394, 2006.

27. "*Model-Based Testing*", MSDN, Web. Dec. 2012. <https://msdn.microsoft.com/en-us/library/ee620469.aspx>.

28. "*List of GUI Testing Tools*" Wikipedia, Wikimedia Foundation, Web. 10 Jan. 2013. <http://en.wikipedia.org/wiki/List_of_GUI_testing_tools>.

29. R. Bryce et al. "*Combinatorial testing*", In *Handbook of Software Engineering Research and Productivity Technologies*, Chapter 14, 2010.

APPENDIX

Gang-of-Four Design Patterns used in this paper

Creational: Builder

The Builder pattern focuses on constructing a complex object step by step. It separates the construction of a complex object from its representation so that the same construction process can create different representations.

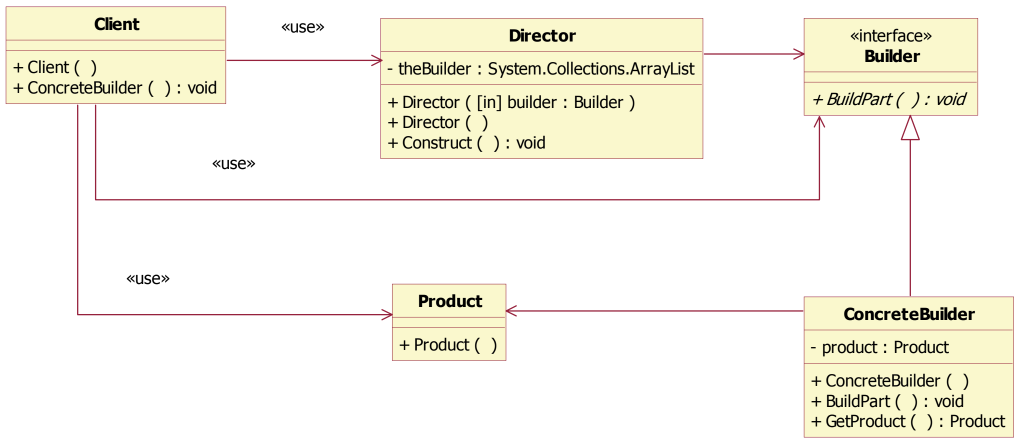


Figure A.1: GoF Creational-Builder Pattern-Class Diagram

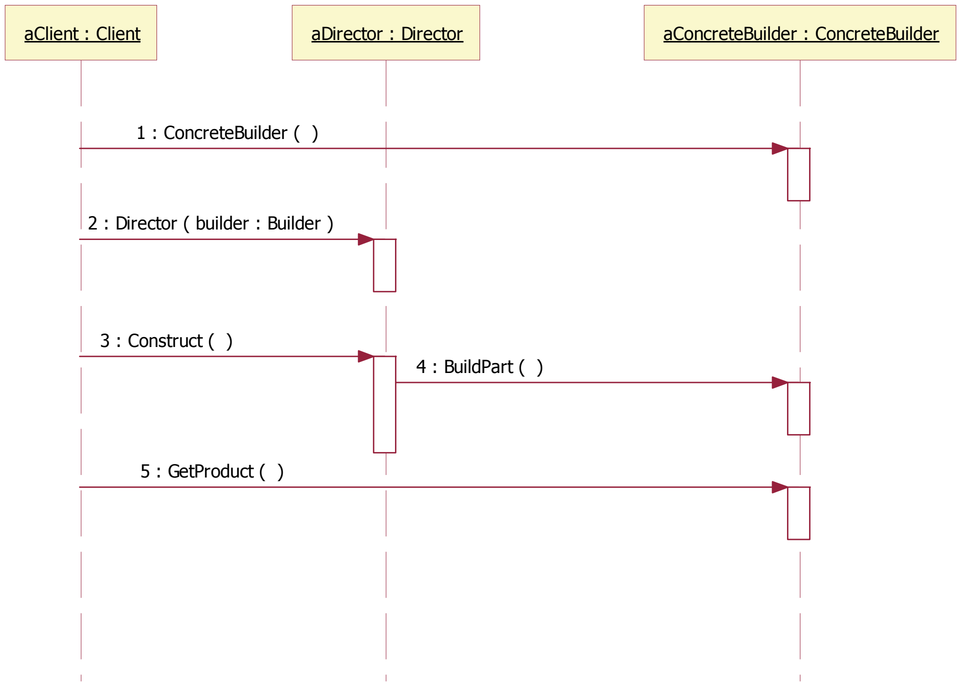


Figure A.2: GoF Creational-Builder Pattern-Sequence Diagram

Creational: Factory Method

The Factory Method defines an interface for creating objects, but lets subclasses decide which classes to instantiate. Although the "new" operator always creates an object, but it fails to encapsulate an object creation. A Factory Method enforces that encapsulation, and allows an object to be requested without inextricable coupling to the act of creation.

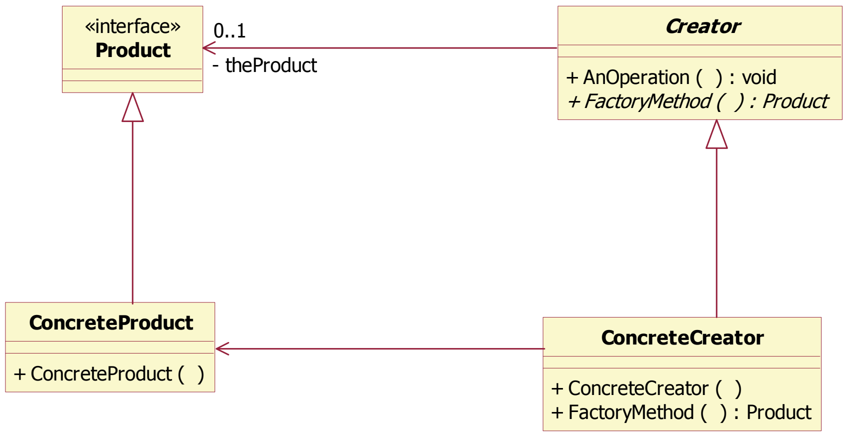


Figure A.3: GoF Creational-Factory Pattern-Class Diagram

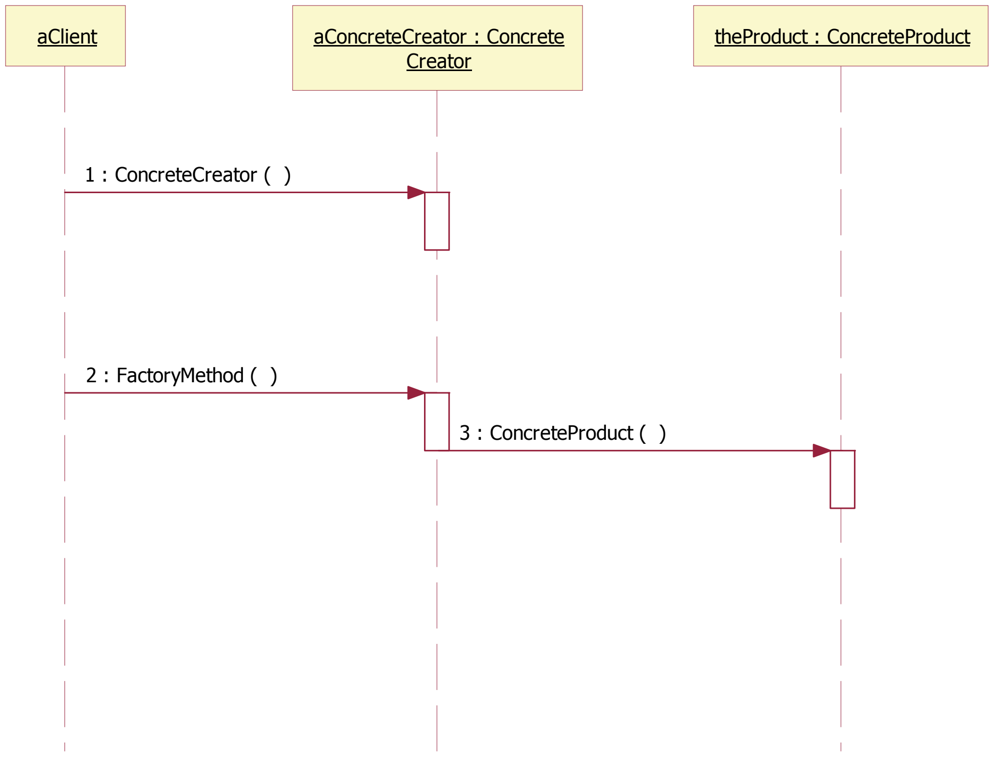


Figure A.4: GoF Creational-Factory Pattern-Sequence Diagram

Creational: Prototype

The Prototype pattern specifies the kinds of objects to create using a prototypical instance. Creation of new objects is done by copying this prototype. It helps avoid expensive new creation by supporting cheap cloning of a pre-initialized prototype.

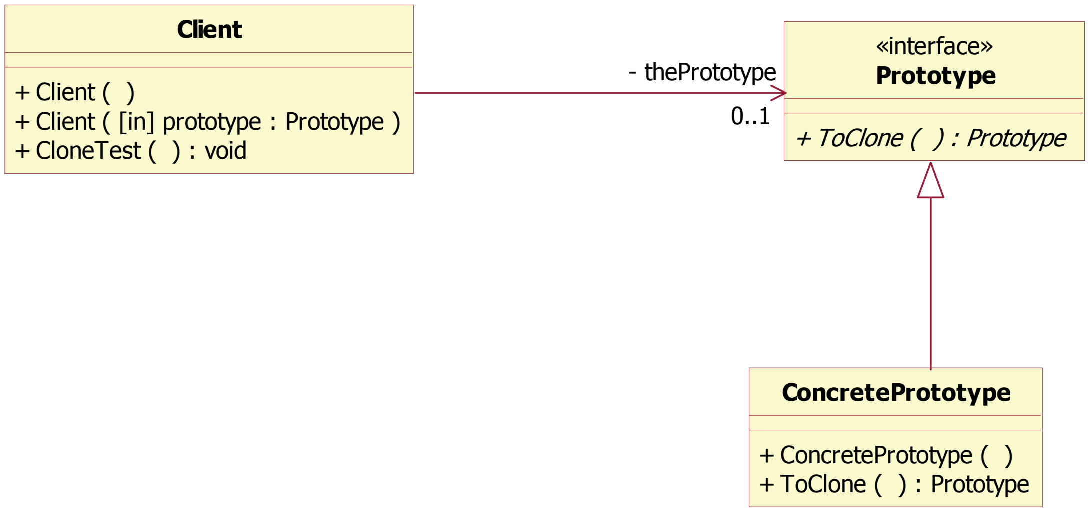


Figure A.5: GoF Creational-Prototype Pattern-Class Diagram

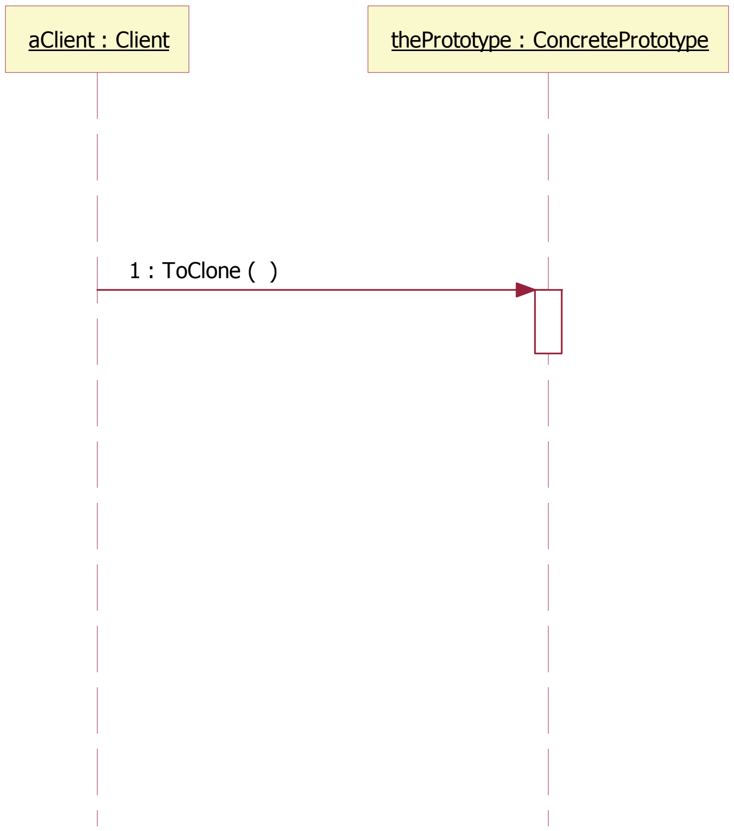


Figure A.6: GoF Creational-Prototype Pattern-Sequence Diagram

Creational: Singleton

The Singleton Pattern ensure that a class has only one instance, and provides a global point of access to it.

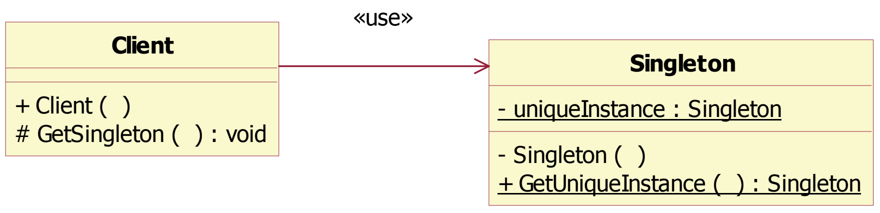

«use»
Client
+ Client ( )
# GetSingleton ( ) : void
Singleton
- uniqueInstance : Singleton
- Singleton ( )
+ GetUniqueInstance ( ) : Singleton

Figure A.7: GoF Creational-Singleton Pattern-Class Diagram

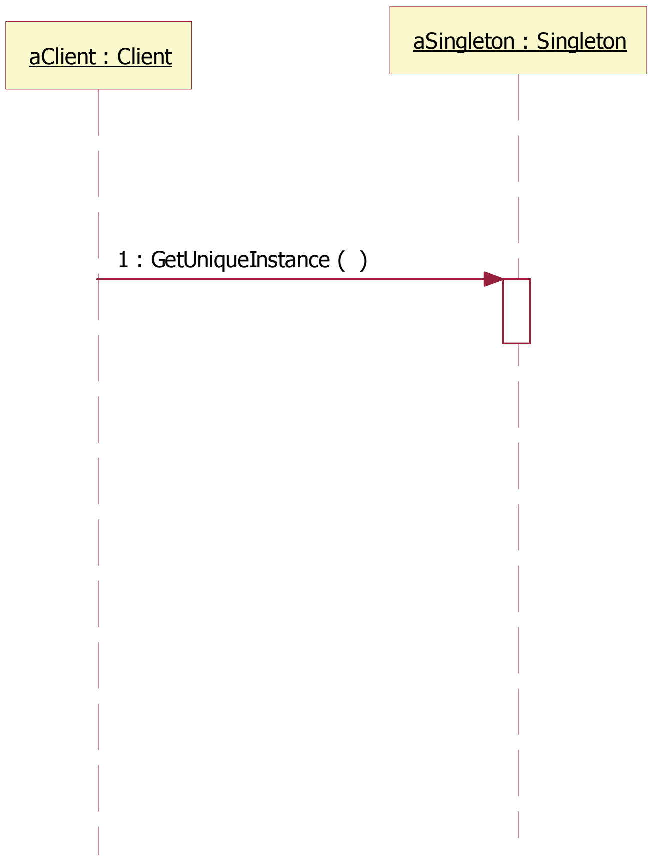


Figure A.8: GoF Creational-Singleton Pattern-Sequence Diagram

Structural: Adapter

The Adapter pattern helps convert an interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.

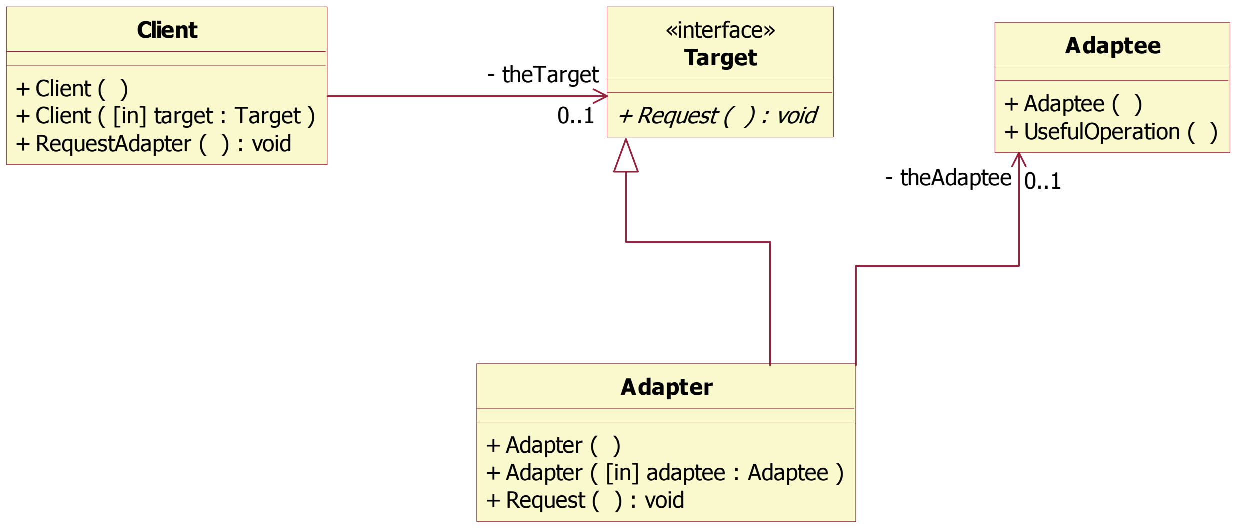


Figure A.9: GoF Structural-Adapter Pattern-Class Diagram

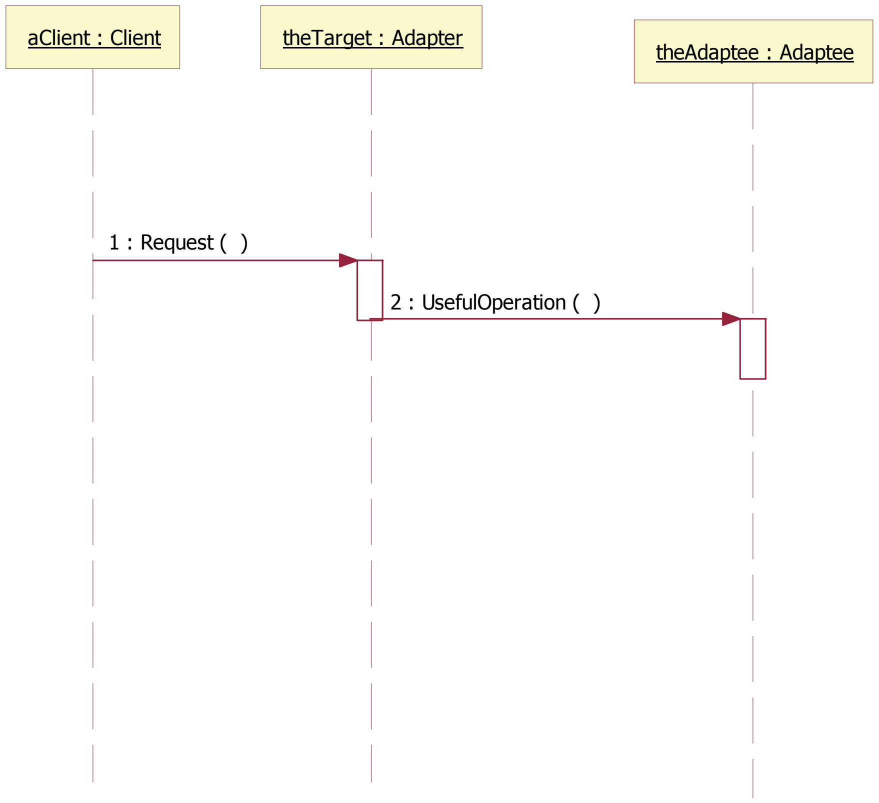


Figure A.10: GoF Structural-Adapter Pattern-Sequence Diagram

Structural: Composite

The Composite pattern helps Compose objects into tree structures to represent whole-part hierarchies. Both individual objects and compositions of objects are treated uniformly.

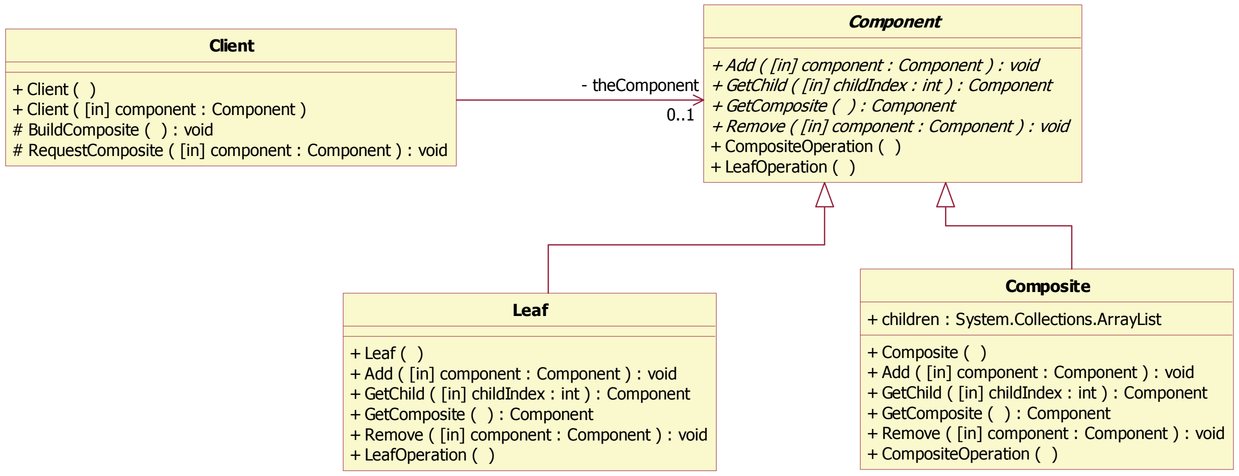

Client
+ Client ( )
+ Client ([in] component : Component)
# BuildComposite ( ) : void
# RequestComposite ([in] component : Component) : void
- theComponent
0..1
Component
+ Add ([in] component : Component) : void
+ GetChild ([in] childIndex : int) : Component
+ GetComposite ( ) : Component
+ Remove ([in] component : Component) : void
+ CompositeOperation ( )
+ LeafOperation ( )
Leaf
+ Leaf ( )
+ Add ([in] component : Component) : void
+ GetChild ([in] childIndex : int) : Component
+ GetComposite ( ) : Component
+ Remove ([in] component : Component) : void
+ LeafOperation ( )
Composite
+ children : System.Collections.ArrayList
+ Composite ( )
+ Add ([in] component : Component) : void
+ GetChild ([in] childIndex : int) : Component
+ GetComposite ( ) : Component
+ Remove ([in] component : Component) : void
+ CompositeOperation ( )

Figure A.11: GoF Structural-Composite Pattern-Class Diagram

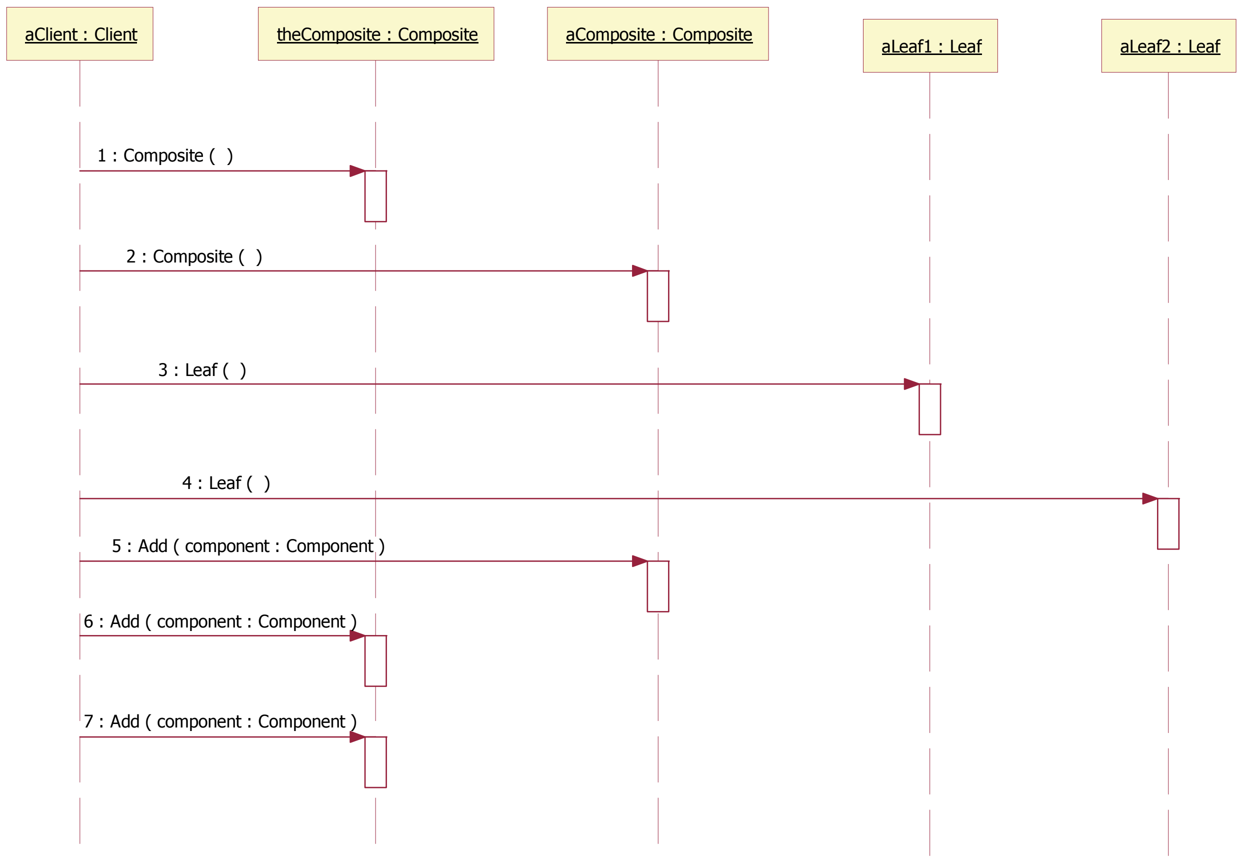


Figure A.12: GoF Structural-Composite Pattern-Sequence Diagram

Structural: Façade

The Façade pattern provides a simplified interface to a single class with a very complex interface. It captures the complexity and collaborations of a component, and delegates to the appropriate methods.

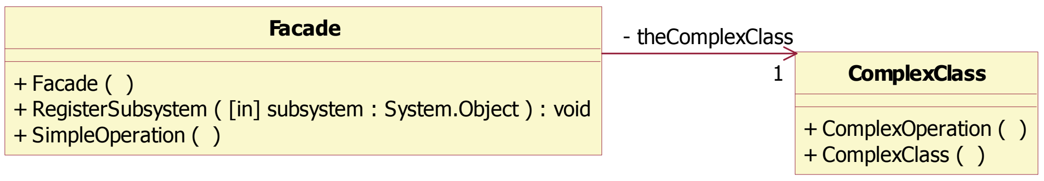


Figure A.13: GoF Structural-Façade Pattern-Class Diagram

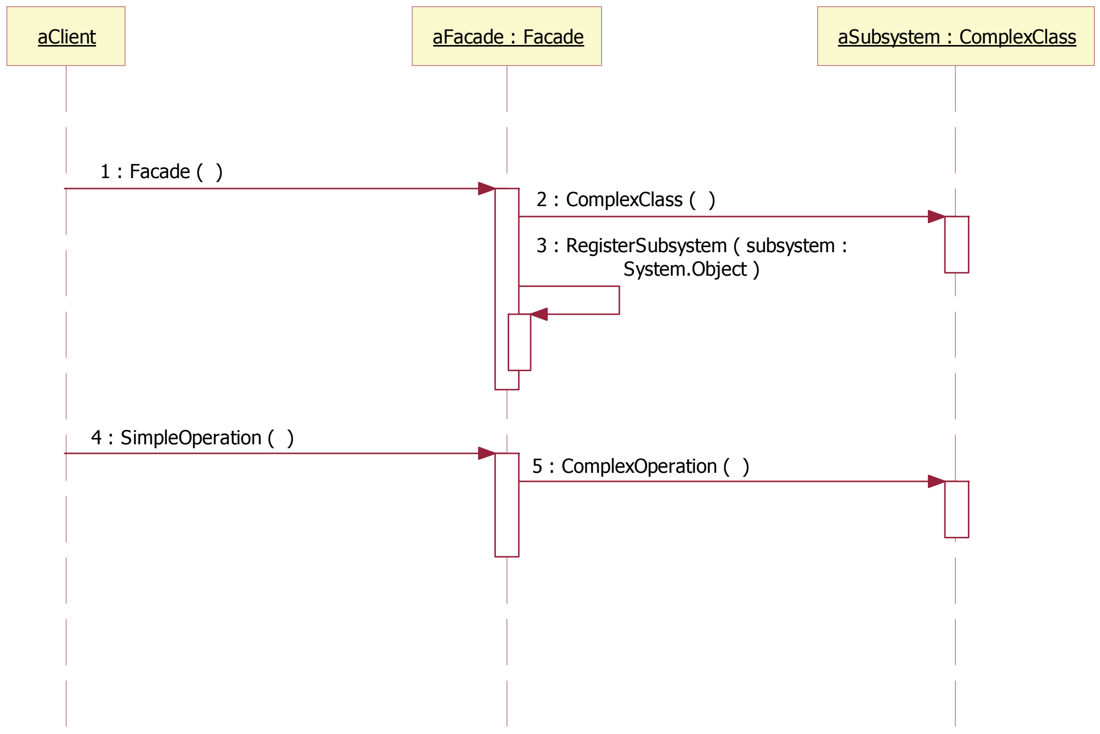


Figure A.14: GoF Structural-Façade Pattern-Sequence Diagram

Structural: Proxy

The Proxy pattern adds a wrapper and delegation to protect the real component from undue complexity. It uses an extra level of indirection but provides the same interface.

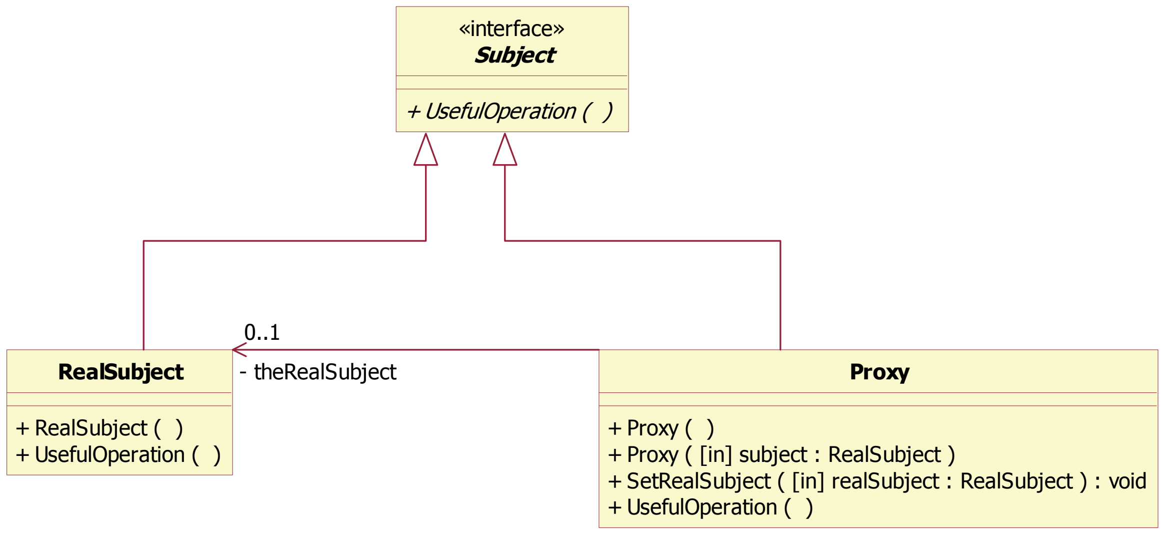


Figure A.15: GoF Structural-Proxy Pattern-Class Diagram

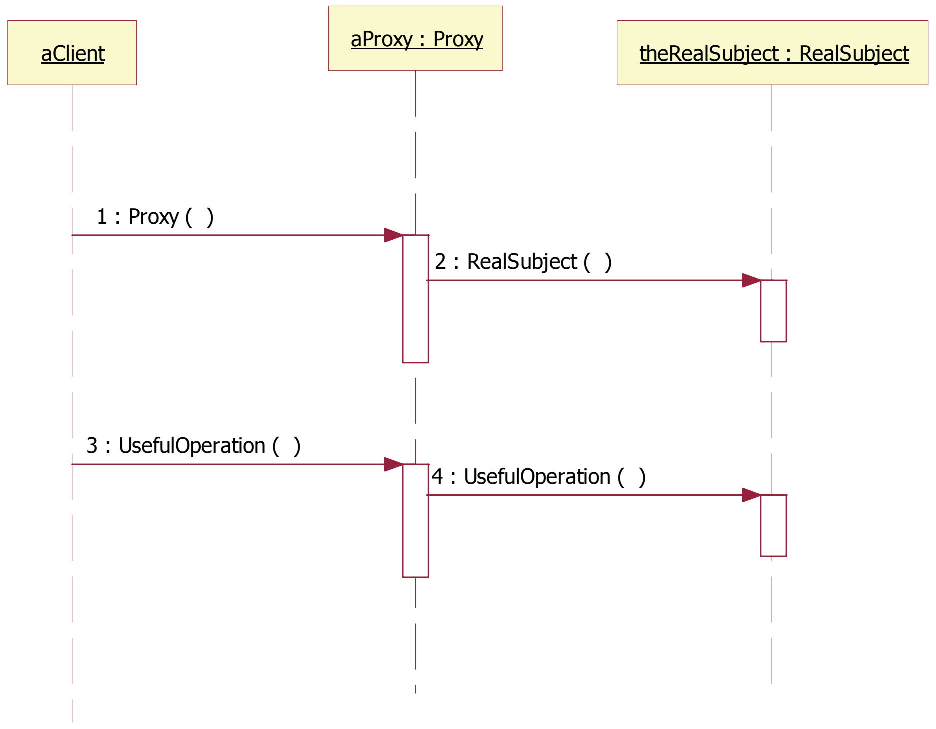


Figure A.16: GoF Structural-Proxy Pattern-Sequence Diagram

Behavioral: Chain of Responsibility

The Chain of Responsibility pattern passes a sender request along a chain of potential receivers. It simplifies coupling, instead of senders and receivers maintaining references to all candidate receivers, each sender keeps a single reference to the head of the chain, and each receiver keeps a single reference to its immediate successor in the chain.

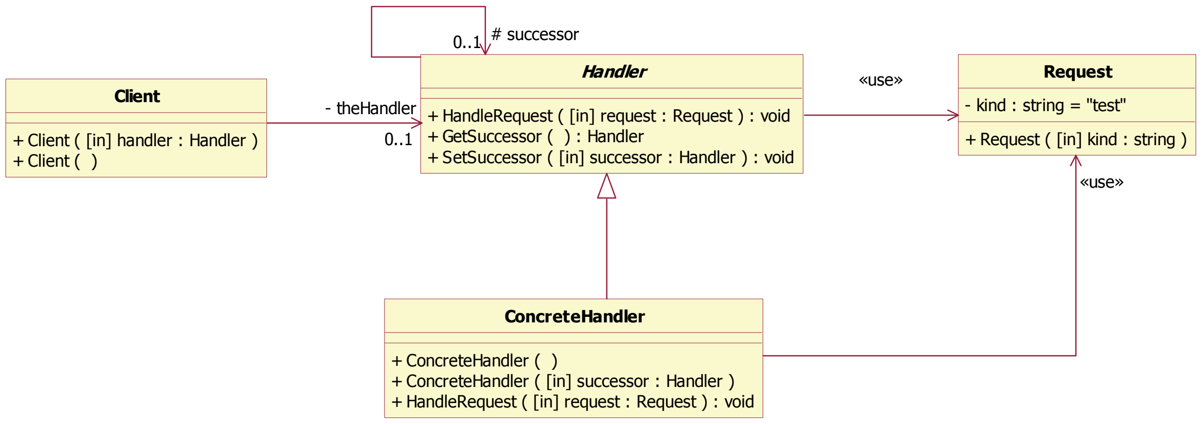

0..1
# successor
Handler
+ HandleRequest ([in] request : Request) : void
+ GetSuccessor ( ) : Handler
+ SetSuccessor ([in] successor : Handler) : void
Client
+ Client ([in] handler : Handler)
+ Client ( )
- theHandler
0..1
«use»
Request
- kind : string = "test"
+ Request ([in] kind : string)
«use»
ConcreteHandler
+ ConcreteHandler ( )
+ ConcreteHandler ([in] successor : Handler)
+ HandleRequest ([in] request : Request) : void

Figure A.17: GoF Behavioral-Chain of Responsibility Pattern-Class Diagram

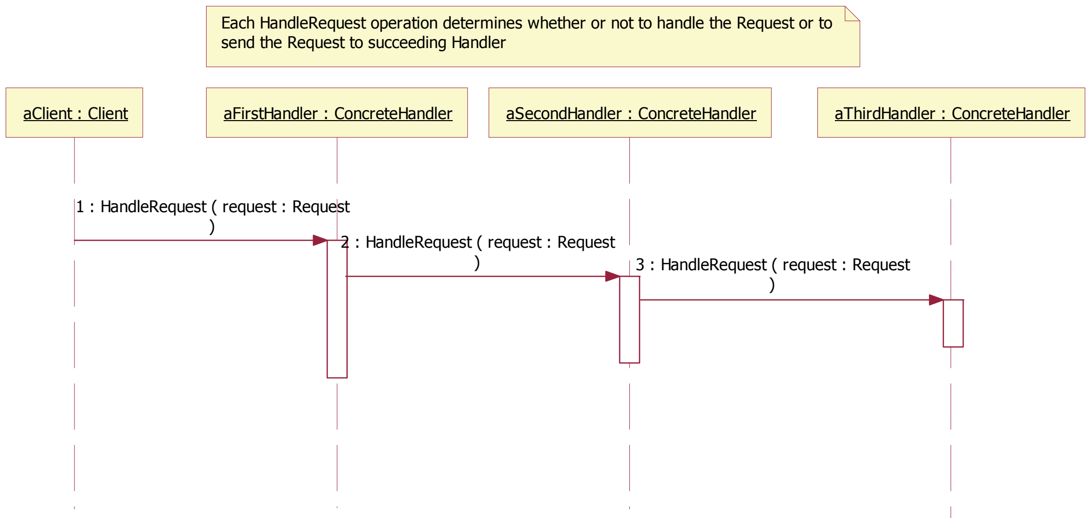

Each HandleRequest operation determines whether or not to handle the Request or to send the Request to succeeding Handler
aClient : Client
aFirstHandler : ConcreteHandler
aSecondHandler : ConcreteHandler
aThirdHandler : ConcreteHandler
1 : HandleRequest (request : Request)
2 : HandleRequest (request : Request)
3 : HandleRequest (request : Request)

Figure A.18: GoF Behavioral-Chain of Responsibility Pattern-Sequence Diagram

Behavioral: Iterator

The Iterator pattern provides ways to access elements in a collection sequentially without exposing the underlying structure of the object.

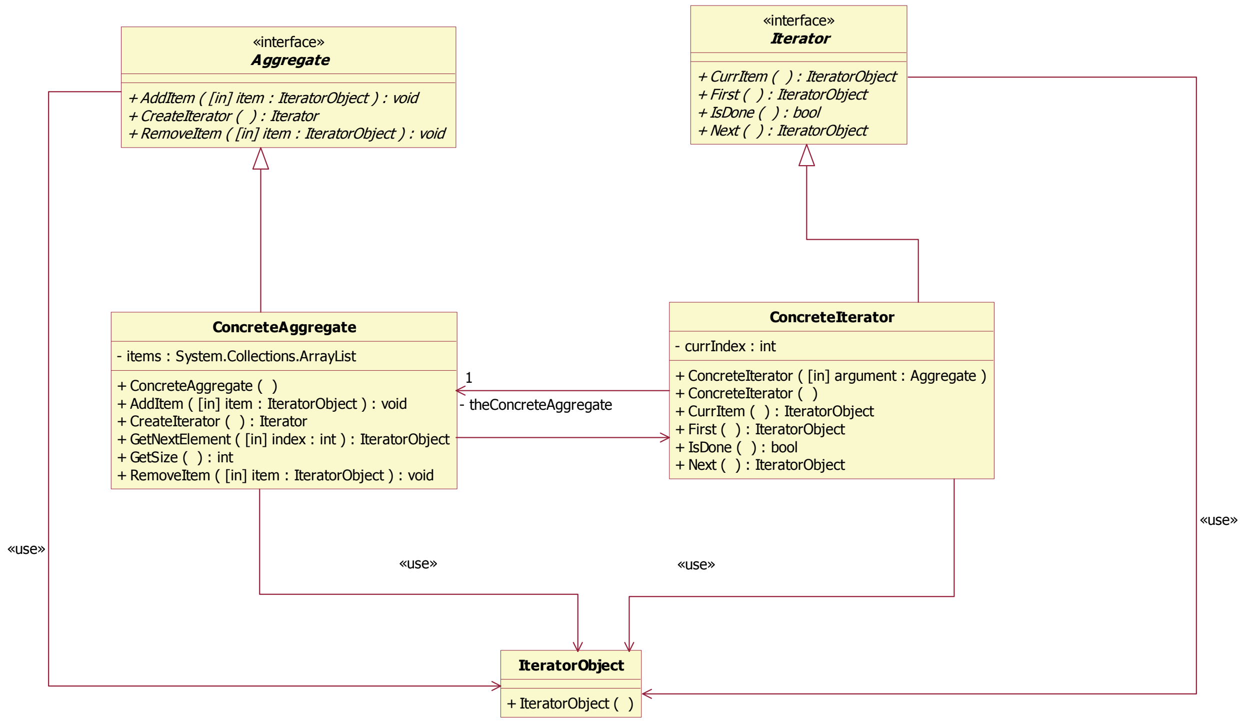

«interface»
Aggregate
+ AddItem ([in] item : IteratorObject) : void
+ CreateIterator ( ) : Iterator
+ RemoveItem ([in] item : IteratorObject) : void
«interface»
Iterator
+ CurrItem ( ) : IteratorObject
+ First ( ) : IteratorObject
+ IsDone ( ) : bool
+ Next ( ) : IteratorObject
ConcreteAggregate
- items : System.Collections.ArrayList
+ ConcreteAggregate ( )
+ AddItem ([in] item : IteratorObject) : void
+ CreateIterator ( ) : Iterator
+ GetNextElement ([in] index : int) : IteratorObject
+ GetSize ( ) : int
+ RemoveItem ([in] item : IteratorObject) : void
1
- theConcreteAggregate
ConcreteIterator
- currIndex : int
+ ConcreteIterator ([in] argument : Aggregate)
+ ConcreteIterator ( )
+ CurrItem ( ) : IteratorObject
+ First ( ) : IteratorObject
+ IsDone ( ) : bool
+ Next ( ) : IteratorObject
«use»
«use»
«use»
«use»
IteratorObject
+ IteratorObject ( )

Figure A.19: GoF Behavioral-Iterator Pattern-Class Diagram

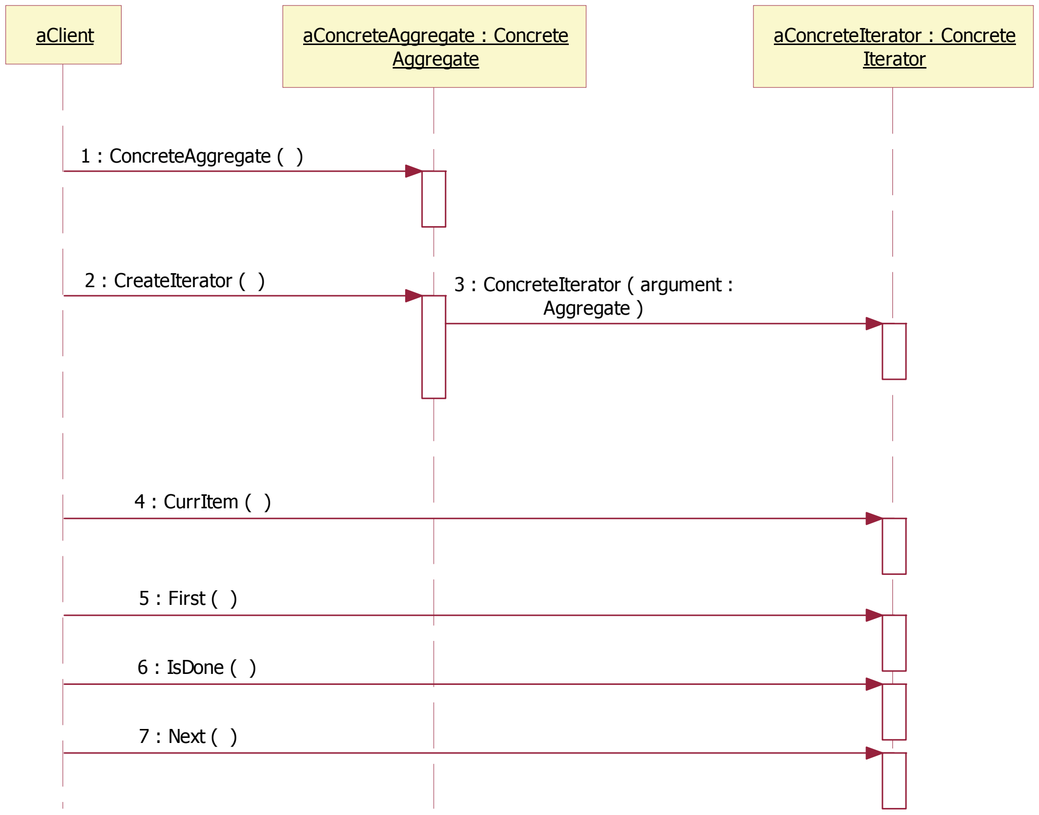

aClient
aConcreteAggregate : Concrete Aggregate
aConcreteIterator : Concrete Iterator
1 : ConcreteAggregate ( )
2 : CreateIterator ( )
3 : ConcreteIterator (argument : Aggregate)
4 : CurrItem ( )
5 : First ( )
6 : IsDone ( )
7 : Next ( )

Figure A.20: GoF Behavioral-Iterator Pattern-Sequence Diagram

Behavioral: Strategy

The Strategy Pattern helps defines a set of algorithms that can be used interchangeably.

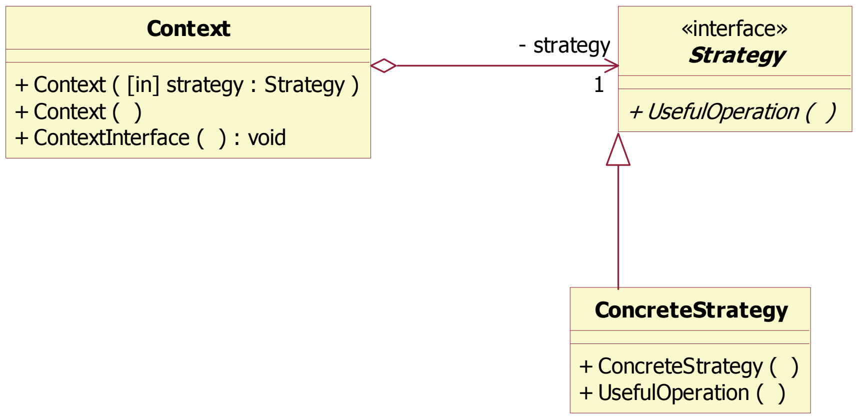


Figure A.21: GoF Behavioral-Strategy Pattern-Class Diagram

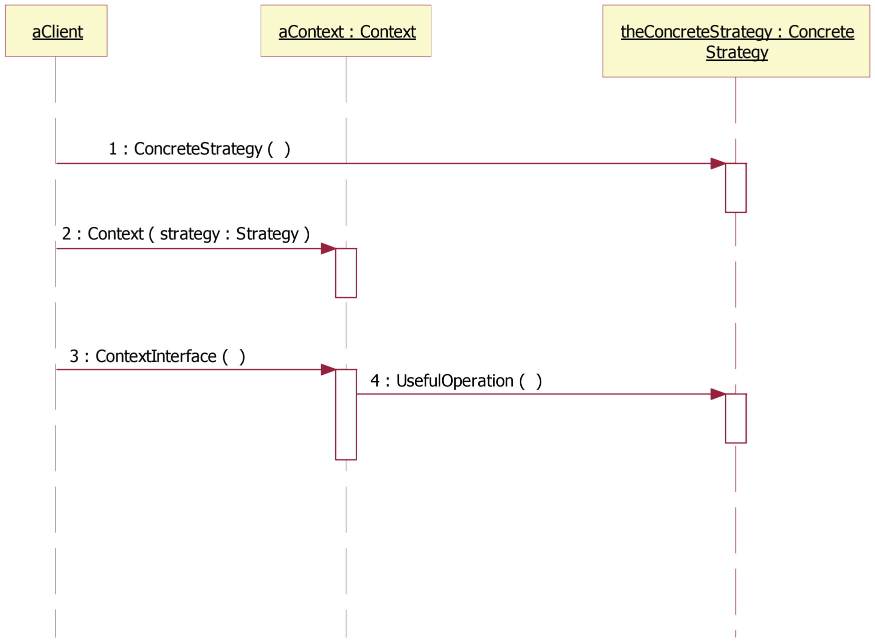


Figure A.22: GoF Behavioral-Strategy Pattern-Sequence Diagram

Behavioral: Template Method

The Template Method pattern helps defines a skeleton of an algorithm in an operation, and defers some steps to derived classes. The invariant steps of the algorithm are implemented in an abstract base class, while the variant steps are either given a default implementation, or no implementation at all.

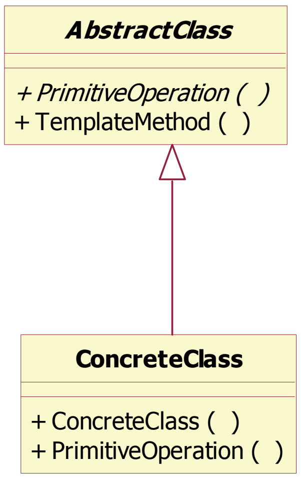


Figure A.23: GoF Behavioral-Template Method Pattern-Class Diagram

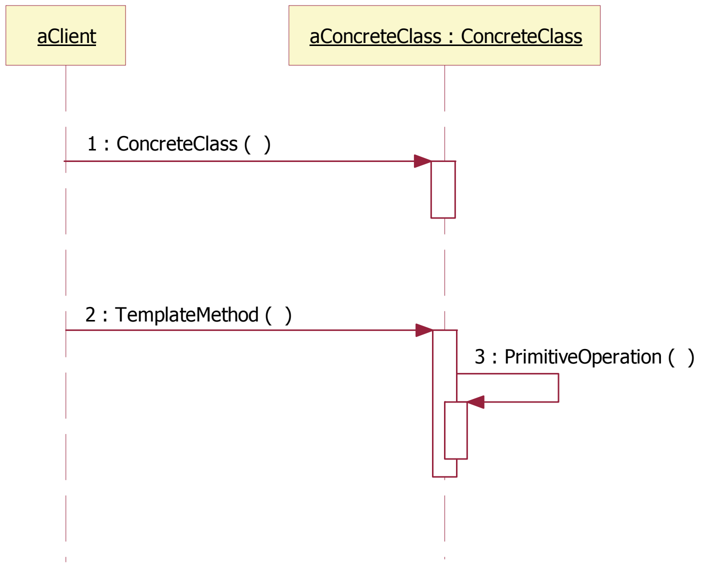


Figure A.24: GoF Behavioral-Template Method Pattern-Sequence Diagram

Miscellaneous Patterns Used

Behavioral: Null Object

The Null Object pattern encapsulates the absence of an object by providing a substitutable alternative that offers suitable default behavior that seamlessly works with existing collaboration.

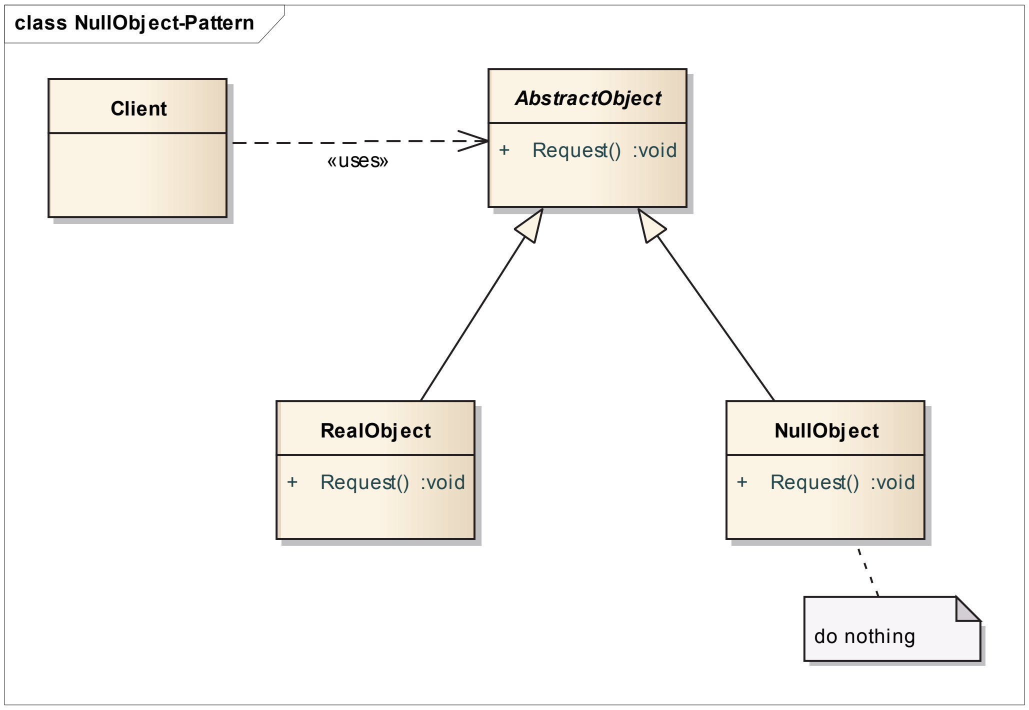


Figure A.25: Behavioral - Null Object