Chapter 13&14 – Multi-layer Design & System Deployment

Tannaz R.Damavandi

Cal Poly Pomona

Outline

- Developing a Multilayer Design
- Design Patterns
- Testing
- Deployment Activities

Overview

- Chapter 12 introduced software design concepts for OO programs, use case realization using the CRC cards technique, and fundamental design principles
- This chapter continues the discussion of OO software design at a more advanced level
- Three layer design is demonstrated using communication diagrams, sequence diagrams, package diagrams and design pattern
- Describe implementation and deployment activities
- Describe four types of software tests and explain how and why each is used
- Implementation includes programming and testing activities. Deployment includes system tests, converting data, training, setting up the production environment, and deploying the solution.

Detailed Design of Multilayer Systems

- CRC Cards focuses on the business logic, also known as problem domain layer of classes
- Three layers include view layer, business logic/problem domain layer, and data access layer
- Questions that come up include
  - How do objects get created in memory?
  - How does the user interface interact with other objects?
  - How are objects handled by the database?
  - Will other objects be necessary?
  - What is the lifespan of each object?

Multilayer System

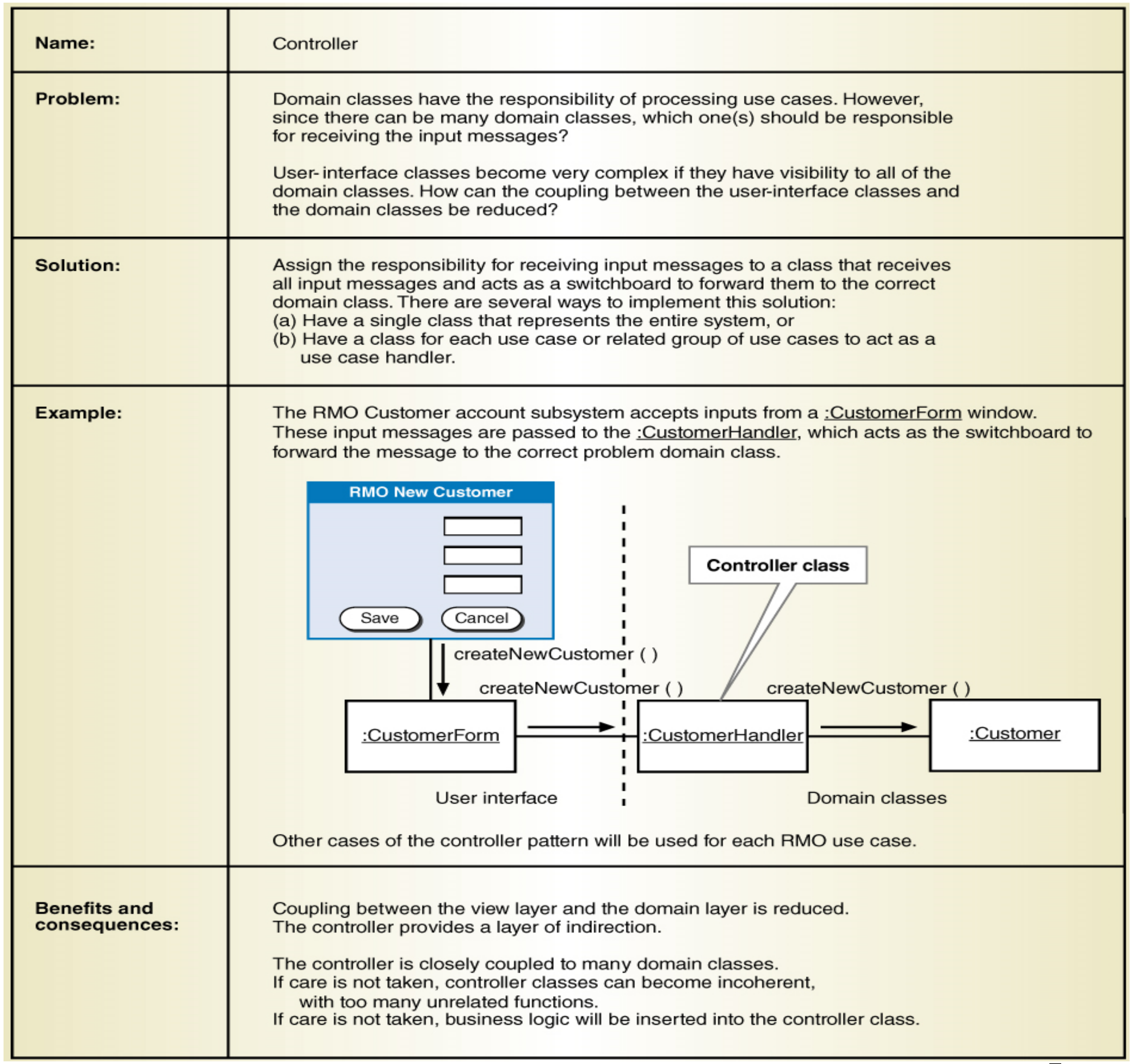
- View Layer Class Responsibilities
  - Display electronic forms and reports.
  - Capture such input events as clicks, rollovers, and key entries.
  - Display data fields.
  - Accept input data.
  - Edit and validate input data.
  - Forward input data to the domain layer classes.
  - Start and shut down the system.
- Domain Layer Class Responsibilities
  - Create problem domain (persistent) classes.
  - Process all business rules with appropriate logic.
  - Prepare persistent classes for storage to the database.
- Data Access Layer Class Responsibilities
  - Establish and maintain connections to the database.
  - Contain all SQL statements.
  - Process result sets (the results of SQL executions) into appropriate domain objects.
  - Disconnect gracefully from the database.

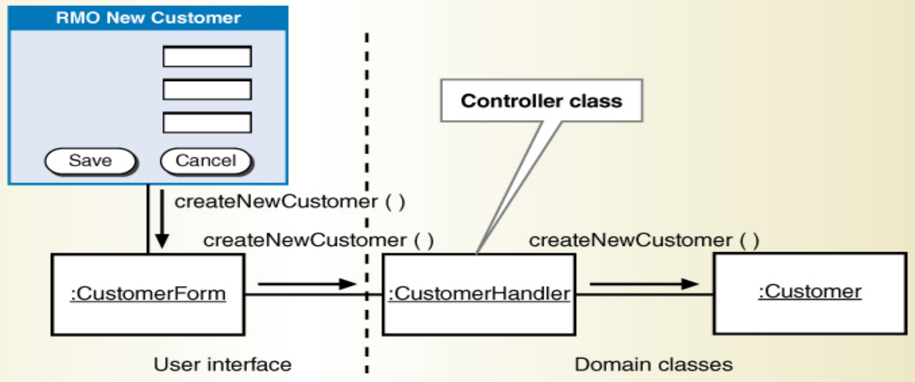
Design Pattern

- Design Pattern—standard design techniques and templates that are widely recognized as good practice
- For common design/coding problems, the design pattern suggests the best way to handle the problem.
- They are written up in design pattern catalogs/references. Include:
  - Pattern name
  - Problem that requires solution
  - The pattern that solves the problem
  - An example of the pattern
  - Benefits and consequences of the a pattern

Controller Pattern

- First step toward multilayer architecture
- Switchboard between user-interface classes and domain layer classes
- Reduces coupling between view and domain layer
- A controller can be created for each use case, however, several controllers can be combined together for a group of related use cases
- It is a completely artificial class – an artifact

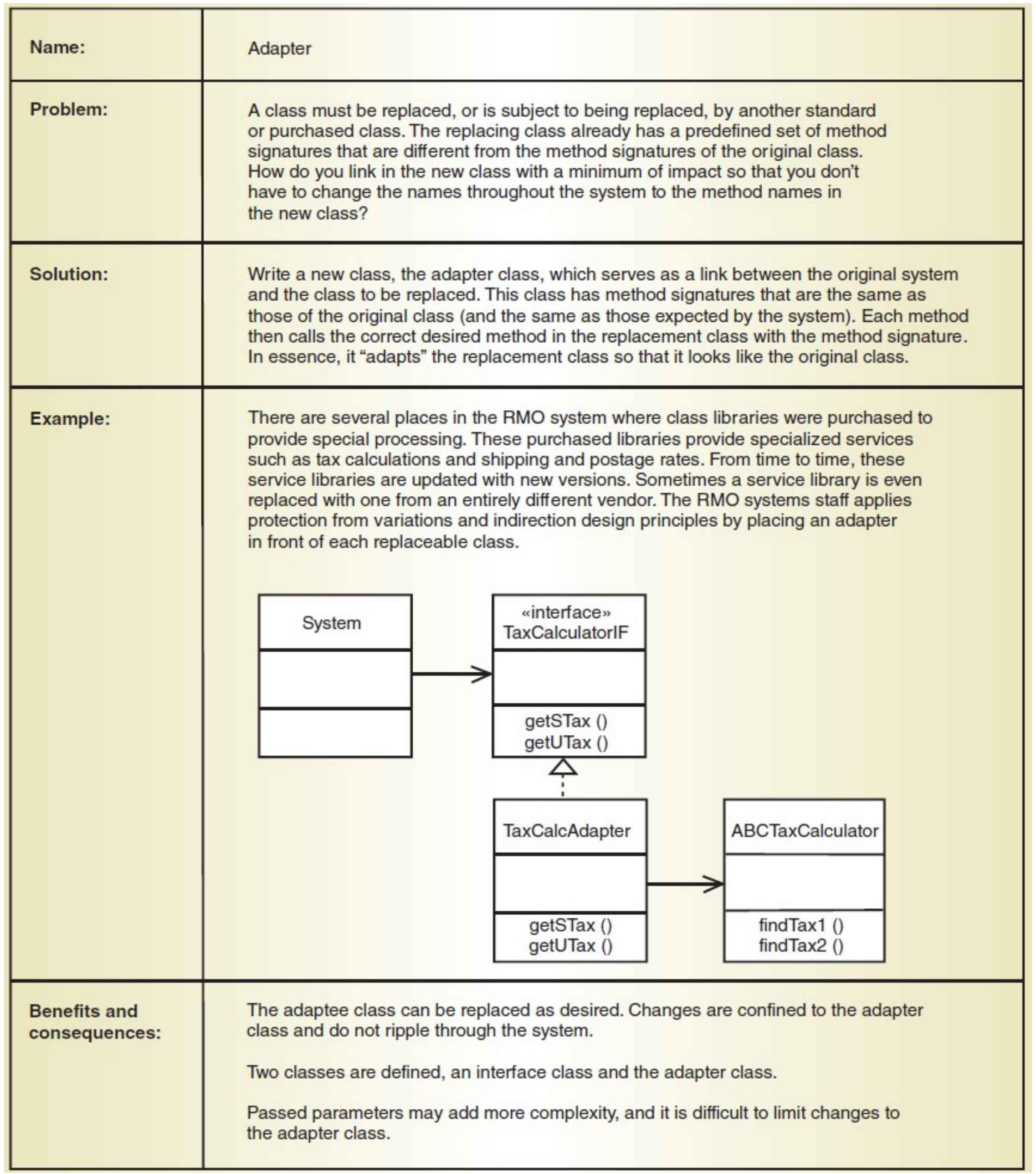


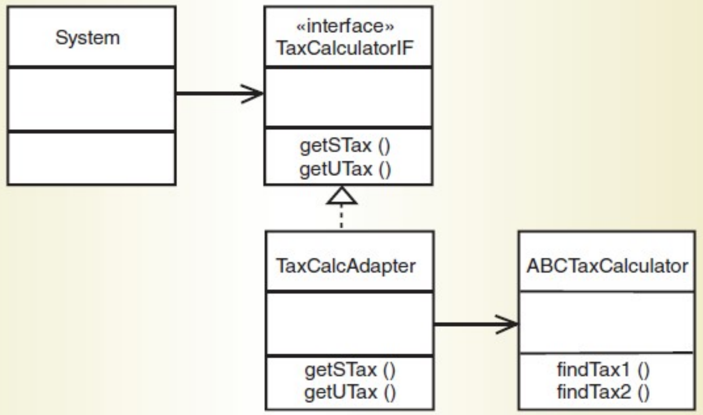
Name:	Controller
Problem:	Domain classes have the responsibility of processing use cases. However, since there can be many domain classes, which one(s) should be responsible for receiving the input messages? User- interface classes become very complex if they have visibility to all of the domain classes. How can the coupling between the user-interface classes and the domain classes be reduced?
Solution:	Assign the responsibility for receiving input messages to a class that receives all input messages and acts as a switchboard to forward them to the correct domain class. There are several ways to implement this solution: (a) Have a single class that represents the entire system, or (b) Have a class for each use case or related group of use cases to act as a use case handler.
Example:	The RMO Customer account subsystem accepts inputs from a :CustomerForm window. These input messages are passed to the :CustomerHandler, which acts as the switchboard to forward the message to the correct problem domain class.  RMO New Customer Save Cancel Controller class createNewCustomer ( ) createNewCustomer ( ) createNewCustomer ( ) :CustomerForm :CustomerHandler :Customer User interface Domain classes Other cases of the controller pattern will be used for each RMO use case.
Benefits and consequences:	Coupling between the view layer and the domain layer is reduced. The controller provides a layer of indirection. The controller is closely coupled to many domain classes. If care is not taken, controller classes can become incoherent, with too many unrelated functions. If care is not taken, business logic will be inserted into the controller class.

More Design Pattern

- Adapter
  - Like an electrical adapter
    - i.e : VGA to HDMI
  - Place an adapter class between your system and an external system
- Factory
  - Use factory class when creation logic is complex for a set of classes
- Singleton
  - Use when only one instance should exist at a time and is shared

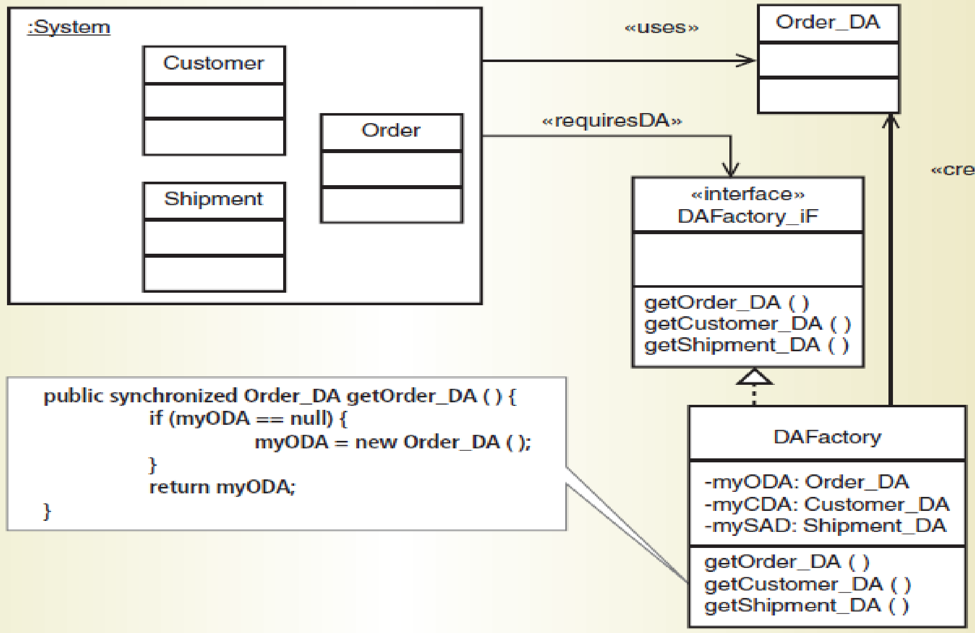
Adapter Design Pattern



Name:	Adapter
Problem:	A class must be replaced, or is subject to being replaced, by another standard or purchased class. The replacing class already has a predefined set of method signatures that are different from the method signatures of the original class. How do you link in the new class with a minimum of impact so that you don't have to change the names throughout the system to the method names in the new class?
Solution:	Write a new class, the adapter class, which serves as a link between the original system and the class to be replaced. This class has method signatures that are the same as those of the original class (and the same as those expected by the system). Each method then calls the correct desired method in the replacement class with the method signature. In essence, it "adapts" the replacement class so that it looks like the original class.
Example:	There are several places in the RMO system where class libraries were purchased to provide special processing. These purchased libraries provide specialized services such as tax calculations and shipping and postage rates. From time to time, these service libraries are updated with new versions. Sometimes a service library is even replaced with one from an entirely different vendor. The RMO systems staff applies protection from variations and indirection design principles by placing an adapter in front of each replaceable class.  System «interface» TaxCalculatorIF getSTax () getUTax () TaxCalcAdapter getSTax () getUTax () ABCTaxCalculator findTax1 () findTax2 ()
Benefits and consequences:	The adaptee class can be replaced as desired. Changes are confined to the adapter class and do not ripple through the system. Two classes are defined, an interface class and the adapter class. Passed parameters may add more complexity, and it is difficult to limit changes to the adapter class.

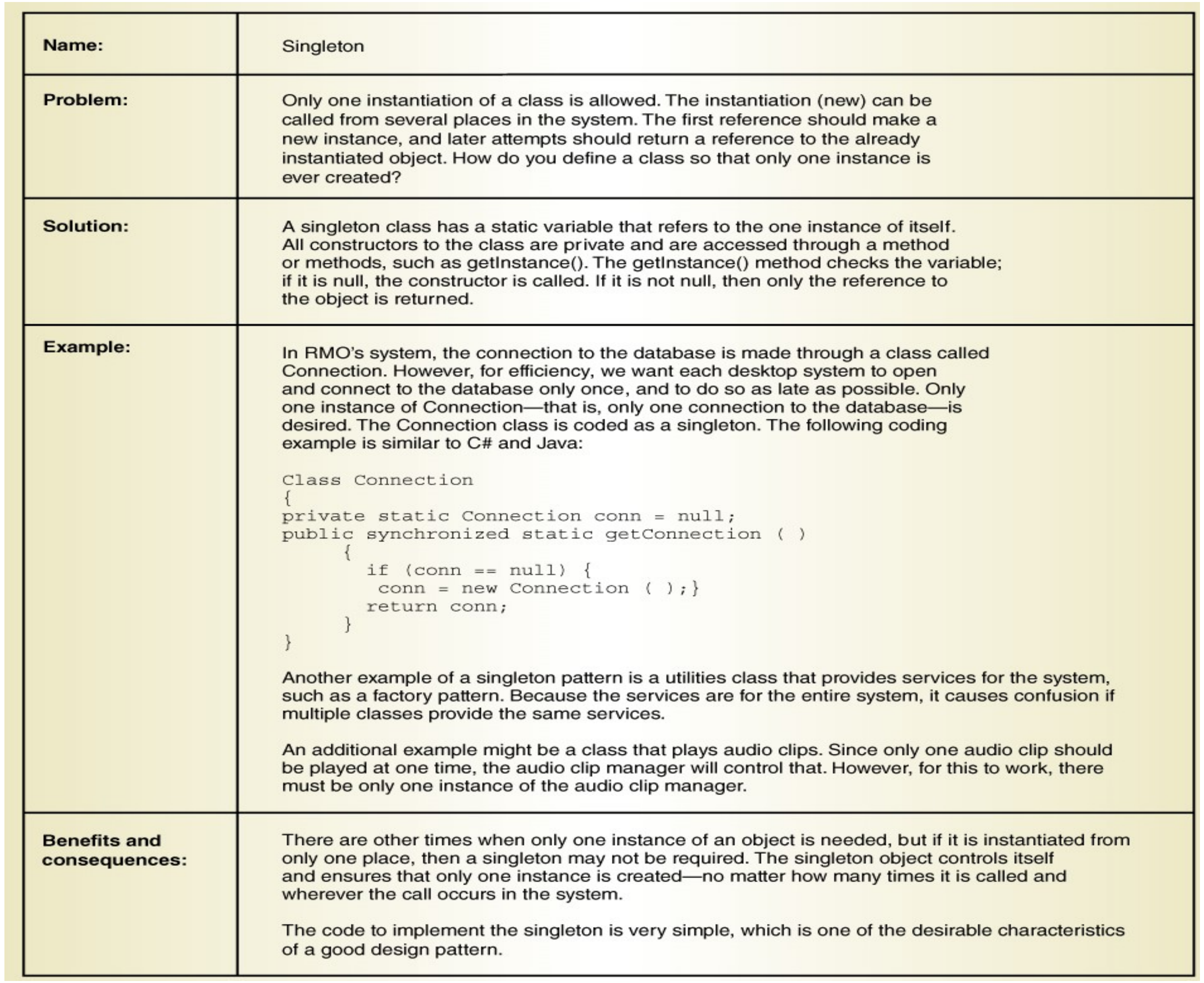
Factory Design Pattern

Name:	Factory or Factory Method
Problem:	Who should be responsible for creating utility type objects that do not specifically belong to the problem domain classes? These utility objects may also be accessed from various places within the system, so a given object may need to be instantiated from several classes.
Solution:	Create an artifact that is a factory class. Its responsibility is only to instantiate utility classes. In many situations, only one instance of a particular utility class is allowed. Hence, all classes that need access to the class come through the factory. The factory ensures that only one instance is created.

Example:	Several places in the RMO system need to get data from an Order object and need to have a reference to an Order_DA [data access] object. The Order_DA object may or may not already have been instantiated. A data access factory is defined and an interface is created. The requesting object uses the methods defined in the interface to request the reference to the Order_DA object. It then can read the database of orders.  :System
	Customer
	Order
	Shipment
	«uses»
	Order_DA
	«requiresDA»
	«creates»
	«interface» DAFactory_iF
	getOrder_DA ( )
	getCustomer_DA ( )
	getShipment_DA ( )
	DAFactory
	-myODA: Order_DA
	-myCDA: Customer_DA
	-mySAD: Shipment_DA
	getOrder_DA ( )
	getCustomer_DA ( )
	getShipment_DA ( )
	`public synchronized Order_DA getOrder_DA ( ) {`
	`if (myODA == null) {`
	`myODA = new Order_DA ( );`
	`}`
	`return myODA;`
	`}`

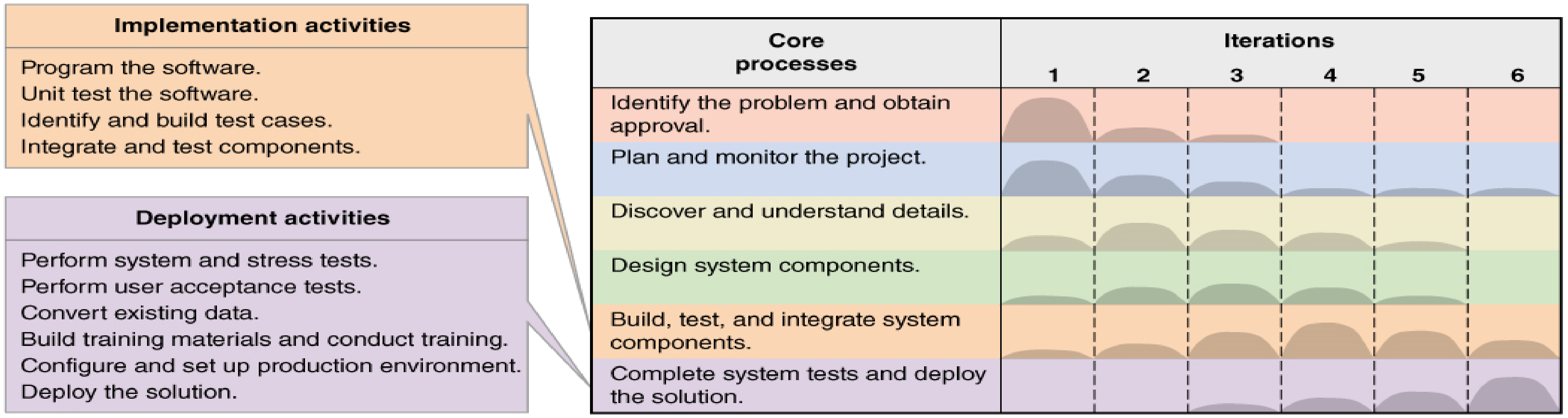
Benefits and Consequences:	Higher cohesion of problem domain classes Less coupling between business logic layer and data layer Smaller, more maintainable classes

Singleton Design Pattern



Name:	Singleton
Problem:	Only one instantiation of a class is allowed. The instantiation (new) can be called from several places in the system. The first reference should make a new instance, and later attempts should return a reference to the already instantiated object. How do you define a class so that only one instance is ever created?
Solution:	A singleton class has a static variable that refers to the one instance of itself. All constructors to the class are private and are accessed through a method or methods, such as getInstance(). The getInstance() method checks the variable; if it is null, the constructor is called. If it is not null, then only the reference to the object is returned.
Example:	In RMO's system, the connection to the database is made through a class called Connection. However, for efficiency, we want each desktop system to open and connect to the database only once, and to do so as late as possible. Only one instance of Connection—that is, only one connection to the database—is desired. The Connection class is coded as a singleton. The following coding example is similar to C# and Java: `Class Connection` `{` `private static Connection conn = null;` `public synchronized static getConnection ( )` `{` `if (conn == null) {` `conn = new Connection ( );}` `return conn;` `}` `}` Another example of a singleton pattern is a utilities class that provides services for the system, such as a factory pattern. Because the services are for the entire system, it causes confusion if multiple classes provide the same services. An additional example might be a class that plays audio clips. Since only one audio clip should be played at one time, the audio clip manager will control that. However, for this to work, there must be only one instance of the audio clip manager.
Benefits and consequences:	There are other times when only one instance of an object is needed, but if it is instantiated from only one place, then a singleton may not be required. The singleton object controls itself and ensures that only one instance is created—no matter how many times it is called and wherever the call occurs in the system. The code to implement the singleton is very simple, which is one of the desirable characteristics of a good design pattern.

Implementation and Deployment Activities



Implementation activities

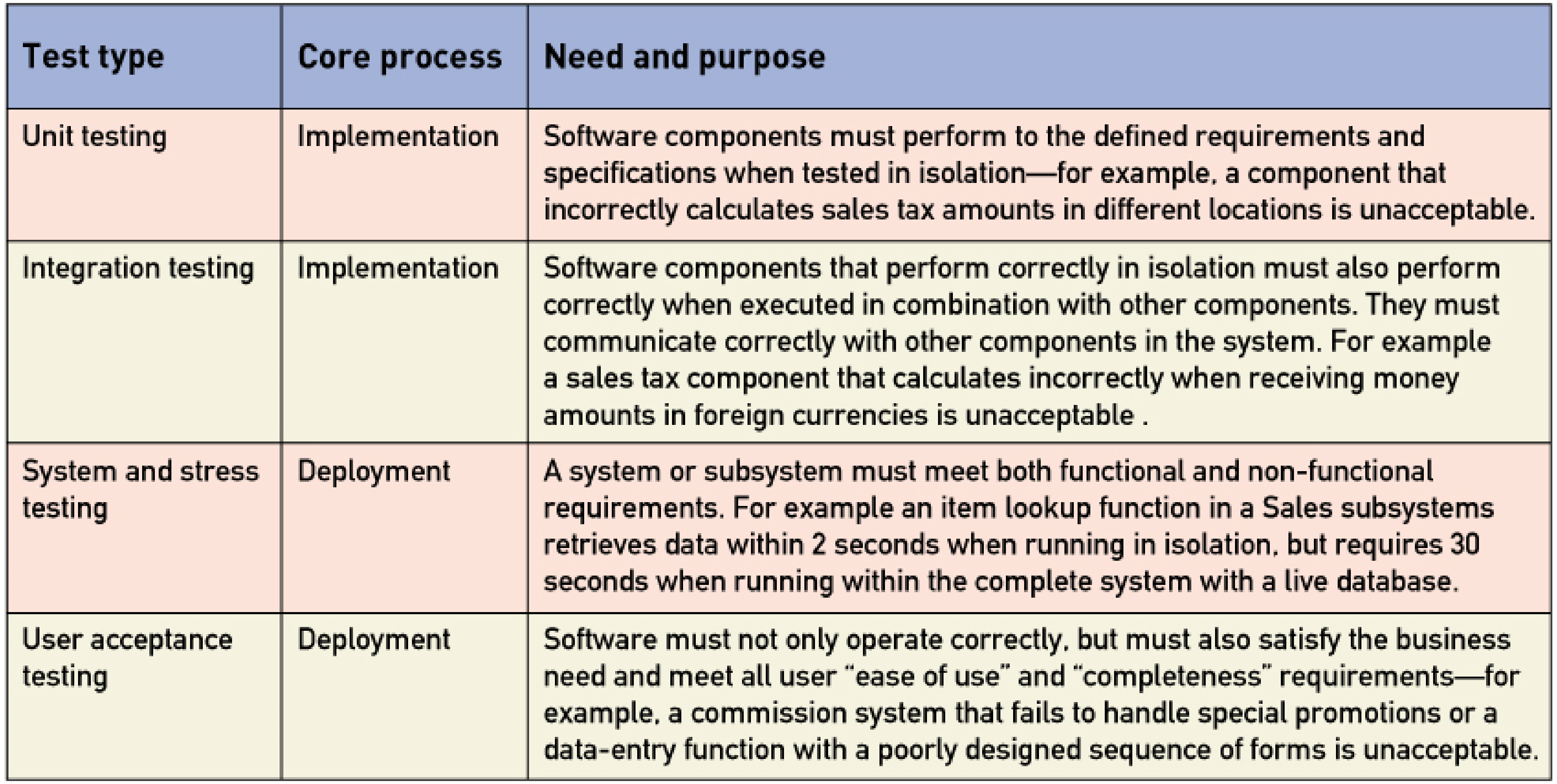
Program the software.
Unit test the software.
Identify and build test cases.
Integrate and test components.

Deployment activities

Perform system and stress tests.
Perform user acceptance tests.
Convert existing data.
Build training materials and conduct training.
Configure and set up production environment.
Deploy the solution.

Core processes	Iterations 1	2	3	4	5	6
Identify the problem and obtain approval.						
Plan and monitor the project.						
Discover and understand details.						
Design system components.						
Build, test, and integrate system components.						
Complete system tests and deploy the solution.						

Most Common Types of Tests



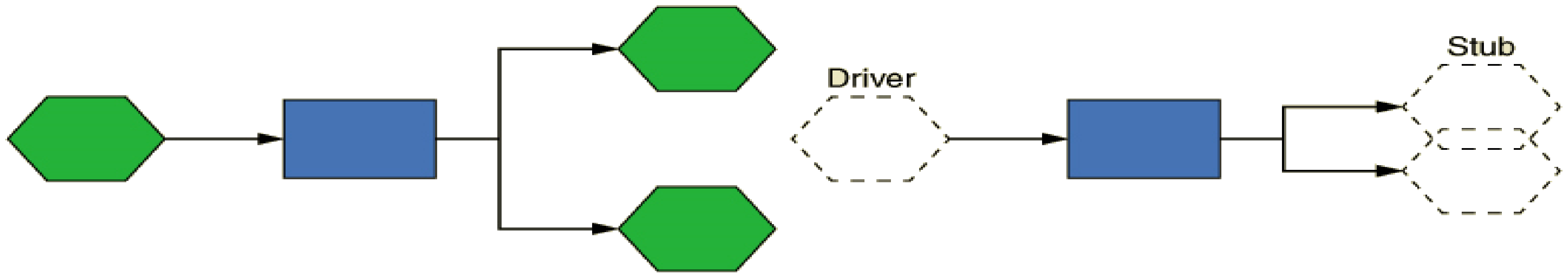
Test type	Core process	Need and purpose
Unit testing	Implementation	Software components must perform to the defined requirements and specifications when tested in isolation—for example, a component that incorrectly calculates sales tax amounts in different locations is unacceptable.
Integration testing	Implementation	Software components that perform correctly in isolation must also perform correctly when executed in combination with other components. They must communicate correctly with other components in the system. For example a sales tax component that calculates incorrectly when receiving money amounts in foreign currencies is unacceptable .
System and stress testing	Deployment	A system or subsystem must meet both functional and non-functional requirements. For example an item lookup function in a Sales subsystems retrieves data within 2 seconds when running in isolation, but requires 30 seconds when running within the complete system with a live database.
User acceptance testing	Deployment	Software must not only operate correctly, but must also satisfy the business need and meet all user “ease of use” and “completeness” requirements—for example, a commission system that fails to handle special promotions or a data-entry function with a poorly designed sequence of forms is unacceptable.

Testing Concept

- Testing – the process of examining a component, subsystem, or system to determine its operational characteristics and whether it contains any defects
- Test case – a formal description of a starting state, one or more events to which the software must respond, and the expected response or ending state
  - Defined based on well understood functional and non-functional requirements
  - Must test all normal and exception situations
- Test data – a set of starting states and events used to test a module, group of modules, or entire system
  - The data that will be used for a test case

Unit Testing

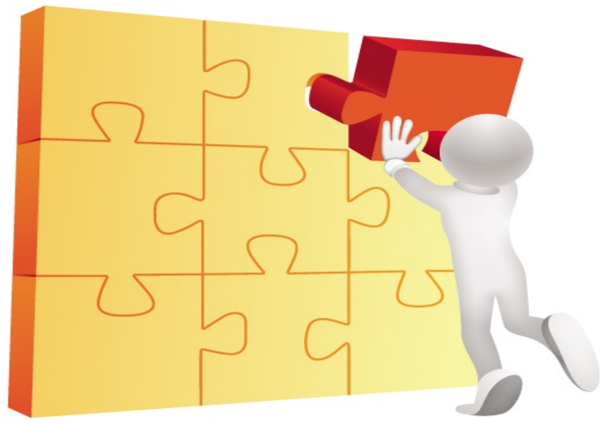
- Unit test – tests of an individual method, class, or component before it is integrated with other software
  - Driver – a method or class developed for unit testing that simulates the behavior of a method that sends a message to the method being tested
  - Stub – a method or class developed for unit testing that simulates the behavior of a method invoked that hasn't yet been written



Driver and Stub Components

Integration Testing

- Integration test – tests of the behavior of a group of methods, classes, or components
  - Interface incompatibility—For example, one method passes a parameter of the wrong data type to another method
  - Parameter values—A method is passed or returns a value that was unexpected, such as a negative number for a price.
  - Run-time exceptions—A method generates an error, such as “out of memory” or “file already in use,” due to conflicting resource needs
  - Unexpected state interactions—The states of two or more objects interact to cause complex failures, as when an OnlineCart class method operates correctly for all possible Customer object states except one

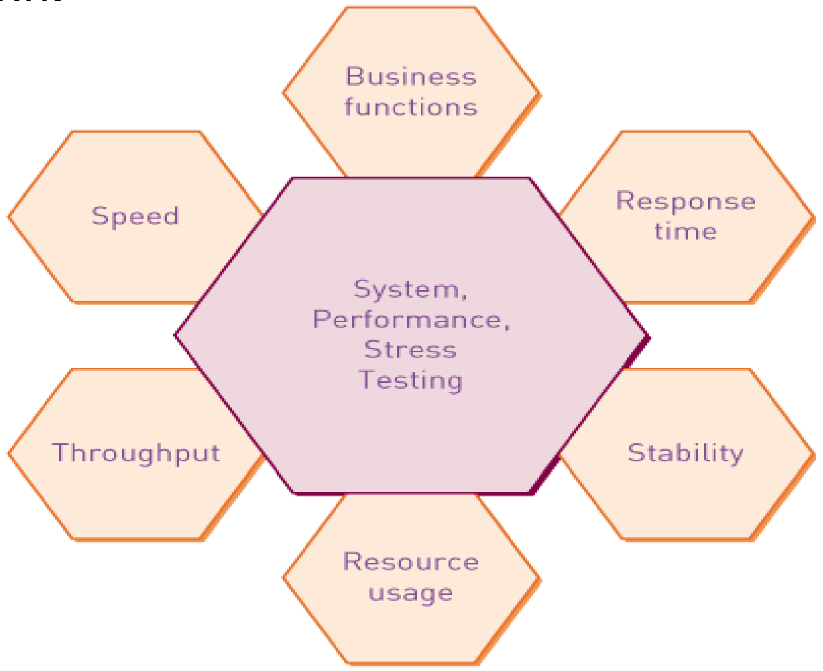


System, Performance, and Stress Testing

- System test – an integration test of an entire system or independent subsystem
  - Can be performed at the end of each iteration
  - Can be performed more frequently
  - Build and smoke test – a system test that is performed daily or several times a week
    - The system is completely compiled and linked (built), and a battery of tests is executed to see whether anything malfunctions in an obvious way (“smokes”)
    - Automated testing tools are used. Catches any problems that may have come up since the last system test

System, Performance, and Stress Testing

- Performance test or stress test – an integration and usability test that determines whether a system or subsystem can meet time-based performance criteria
  - Response time – the desired or maximum allowable time limit for software response to a query or update
  - Throughput – the desired or minimum number of queries and transactions that must be processed per minute or h



User Acceptance Testing (UAT)

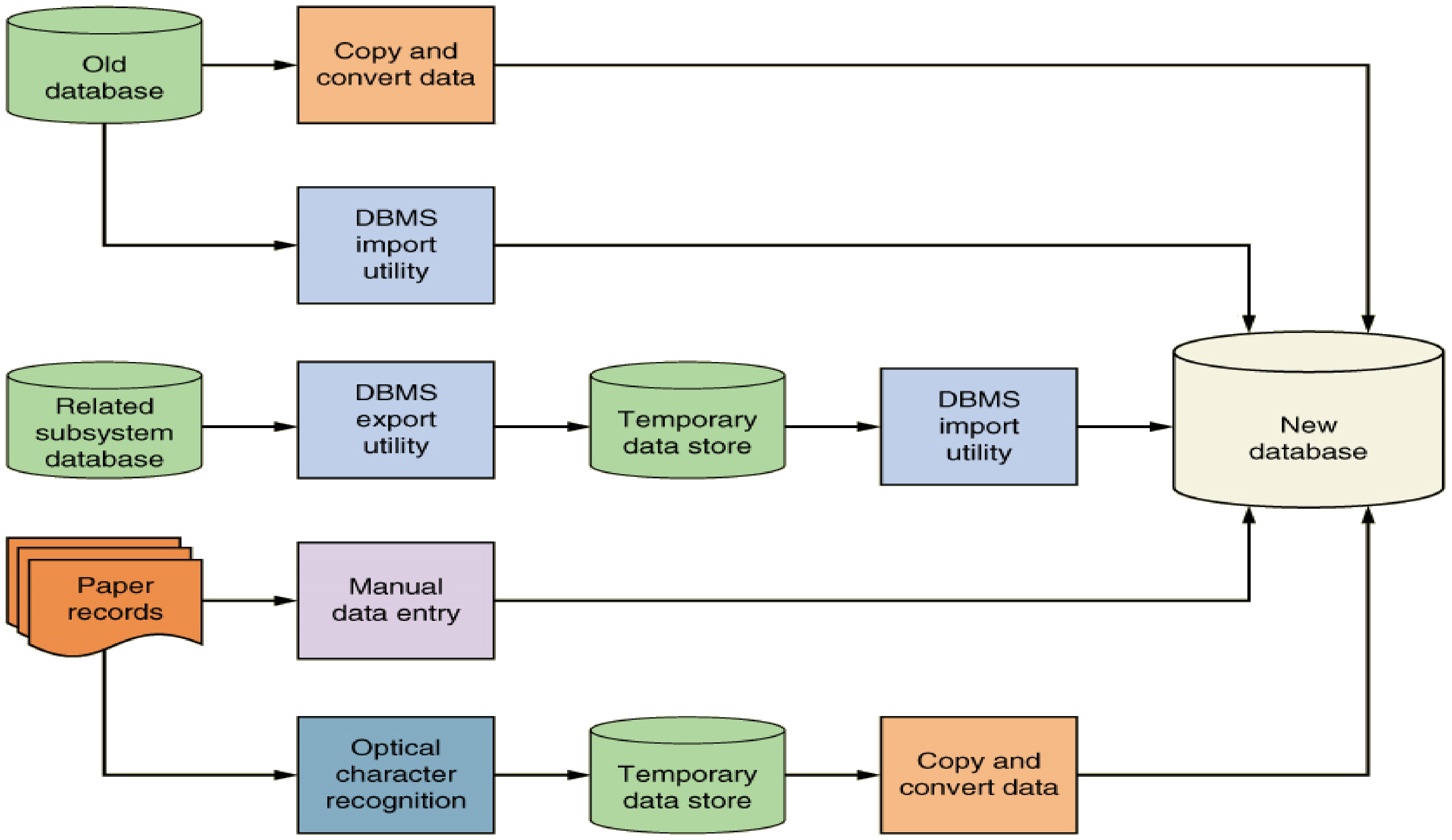
- User acceptance test
  - a system test performed to determine whether the system fulfills user requirements
- May be performed near the end of the project (or at end of later project iterations)
- A very formal activity in most development projects. Payments tied to passing tests
- Details of acceptance tests are sometimes included in the request for proposal (RFP) and procurement contract

Converting and Initializing Data

- An operational system requires a fully populated database to support ongoing processing
- Data needed at system startup can be obtained from these sources:
  - Files or databases of a system being replaced
  - Manual records
  - Files or databases from other systems in the organization
  - User feedback during normal system operation
- Reuse existing databases
  - Modify or update existing data
- Reload databases
  - Copy and convert the data
  - Export and import data from distinct DBMSs
  - Data entry from paper documents

Converting and Initializing Data

- Complex data conversion example



Training Users

- Training is needed for end users and system operators
- Training for end users must emphasize hands-on use for specific business processes or functions, such as order entry, inventory control, or accounting
  - Widely varying skill and experience levels call for at least some hands-on training, including practice exercises, questions and answers, and one-on-one tutorials
- System operator training can be much less formal when the operators aren't end users
  - Experienced computer operators and administrators can learn most or all they need to know by self-study
- System Documentation
  - Descriptions of system requirements and architecture to help maintenance and upgrade of the system
- User Documentation
  - How to interact with and use the system for end users and system operators

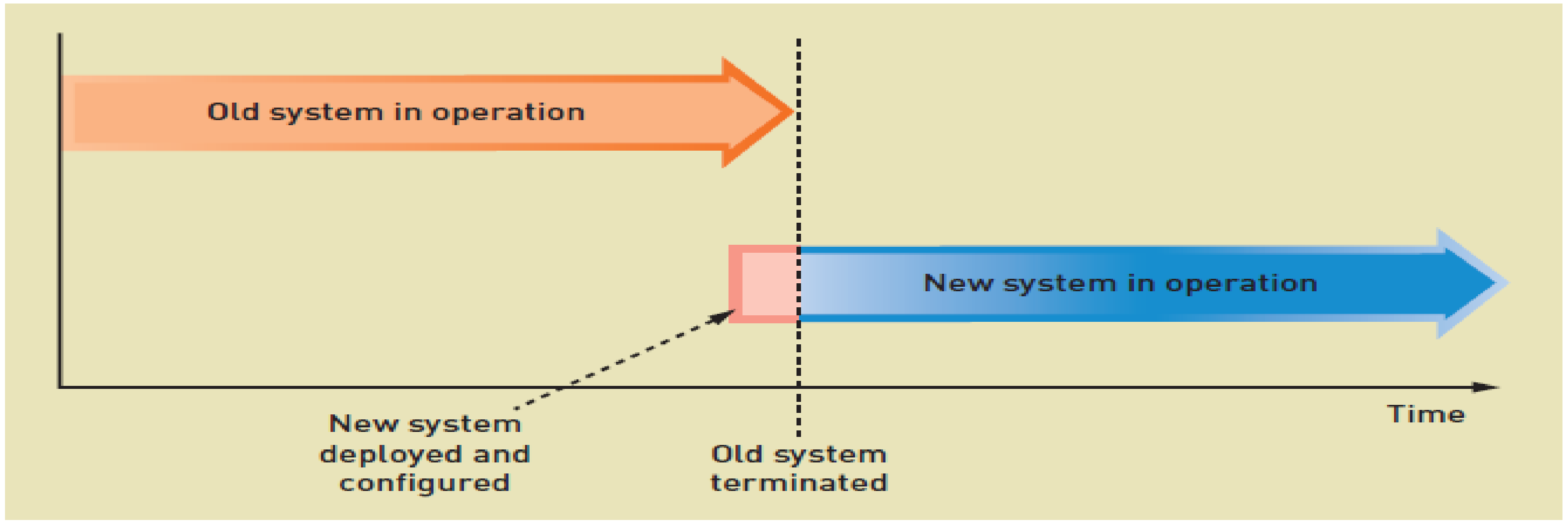
Planning and Managing- Implementation, Testing and Deployment

Different approaches

- Direct deployment
- Parallel deployment
- Phased deployment

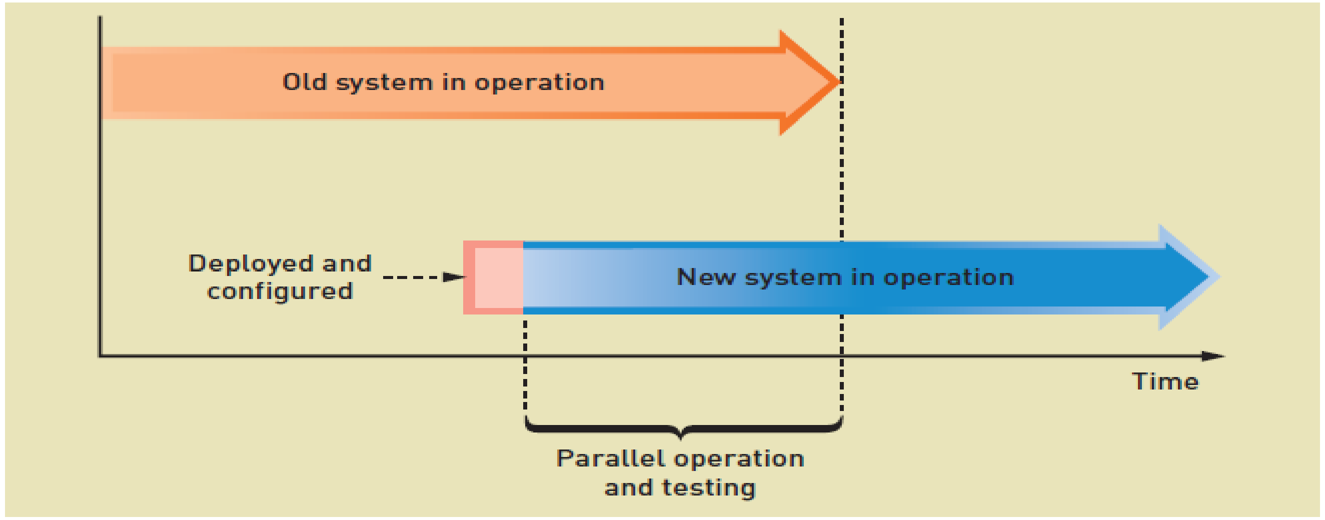
Planning and Managing- Implementation, Testing and Deployment

- Direct deployment – a deployment method that installs a new system, quickly makes it operational, and immediately turns off any overlapping systems
  - Higher risk, lower cost



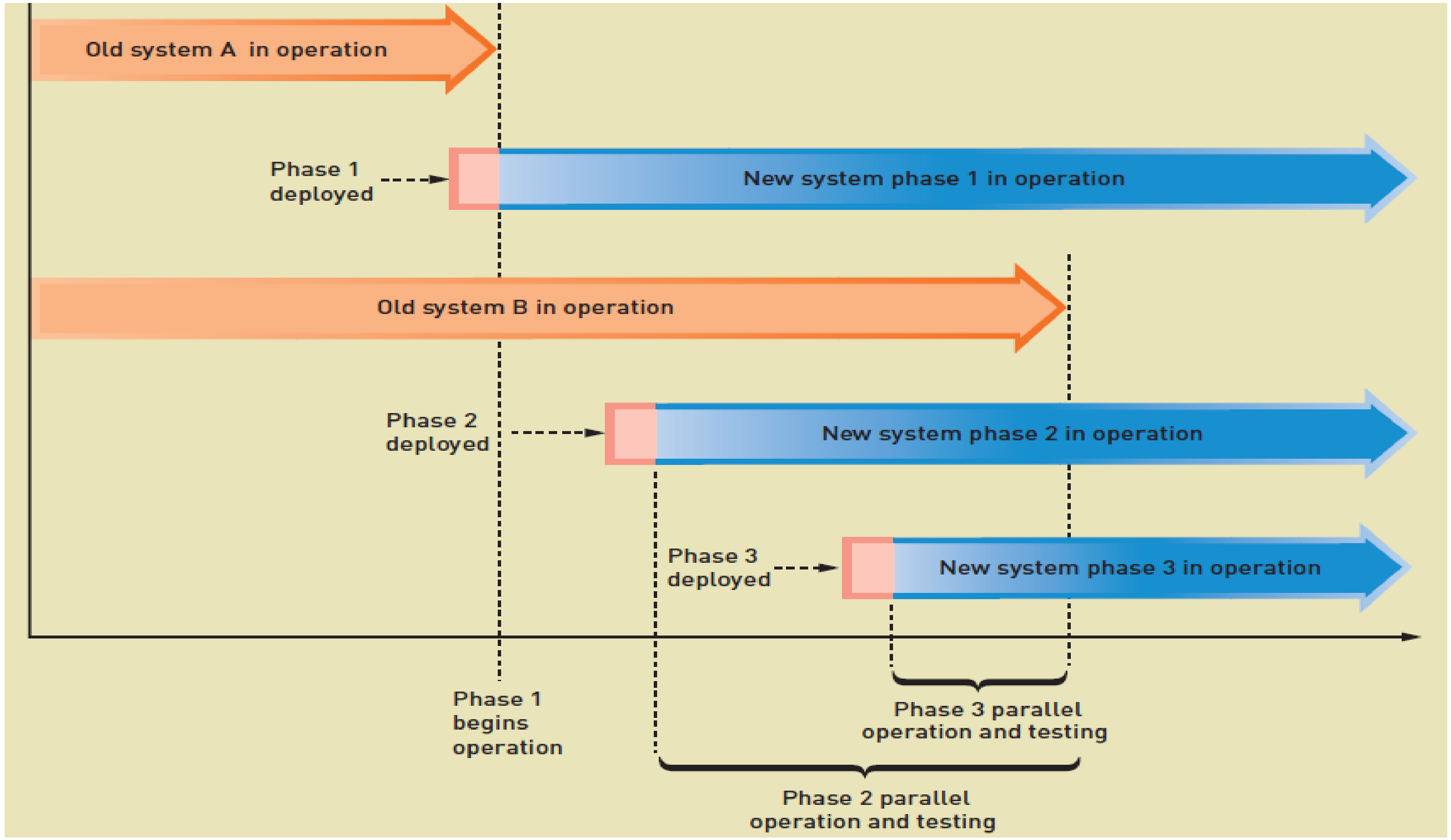
Planning and Managing- Implementation, Testing and Deployment

- Parallel deployment – a deployment method that operates the old and the new systems for an extended time period
  - Lower risk, higher cost
  - Parallel deployment – Problems
    - Incompatible inputs (old and new)
    - Heavier load on equipment, may not have sufficient capacity for both
    - Heavier load on staff, may require overtime
  - Partial parallel may be an option
    - Process only a subset of the data
    - Use only part of the system – only some functions



Planning and Managing- Implementation, Testing and Deployment

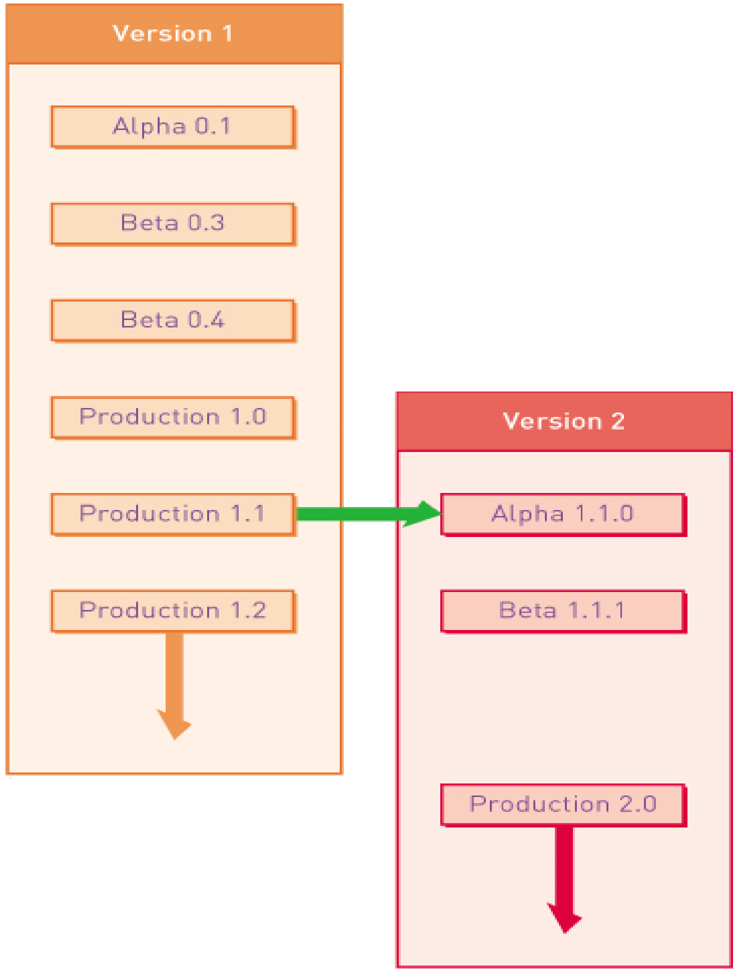
- Phased deployment –a deployment method that installs a new system and makes it operational in a series of steps or phases


Old system A in operation
Phase 1 deployed
New system phase 1 in operation
Old system B in operation
Phase 2 deployed
New system phase 2 in operation
Phase 3 deployed
New system phase 3 in operation
Phase 1 begins operation
Phase 3 parallel operation and testing
Phase 2 parallel operation and testing

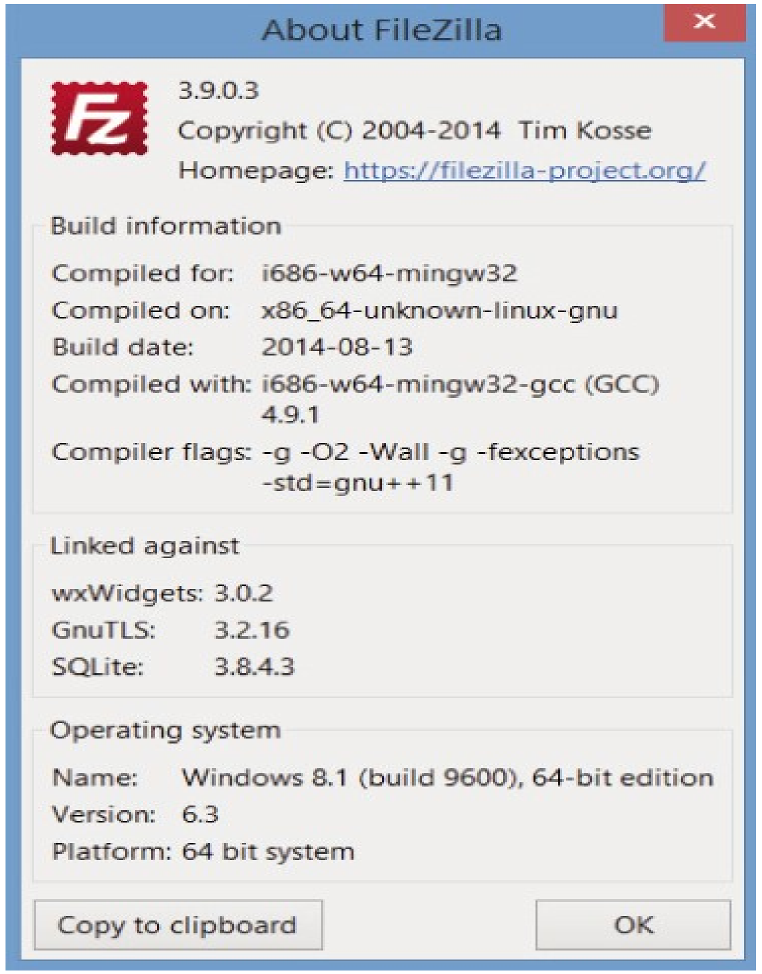
Planning and Managing- Implementation, Testing and Deployment

- Change and Version Control – tools and processes handle the complexity associated with testing and supporting a system through multiple versions
  - Alpha version – a test version that is incomplete but ready for some level of rigorous integration or usability testing
  - Beta version – a test version that is stable enough to be tested by end users over an extended period of time
  - Production version, release version, or production release – a system version that is formally distributed to users or made operational for long-term use
  - Maintenance release – a system update that provides bug fixes and small changes to existing features

Version Control



Need for version control



About FileZilla

3.9.0.3
Copyright (C) 2004-2014 Tim Kosse
Homepage: https://filezilla-project.org/

Build information

Compiled for:	i686-w64-mingw32
Compiled on:	x86_64-unknown-linux-gnu
Build date:	2014-08-13
Compiled with:	i686-w64-mingw32-gcc (GCC) 4.9.1
Compiler flags:	-g -O2 -Wall -g -fexceptions -std=gnu++11

Linked against

wxWidgets:	3.0.2
GnuTLS:	3.2.16
SQLite:	3.8.4.3

Operating system

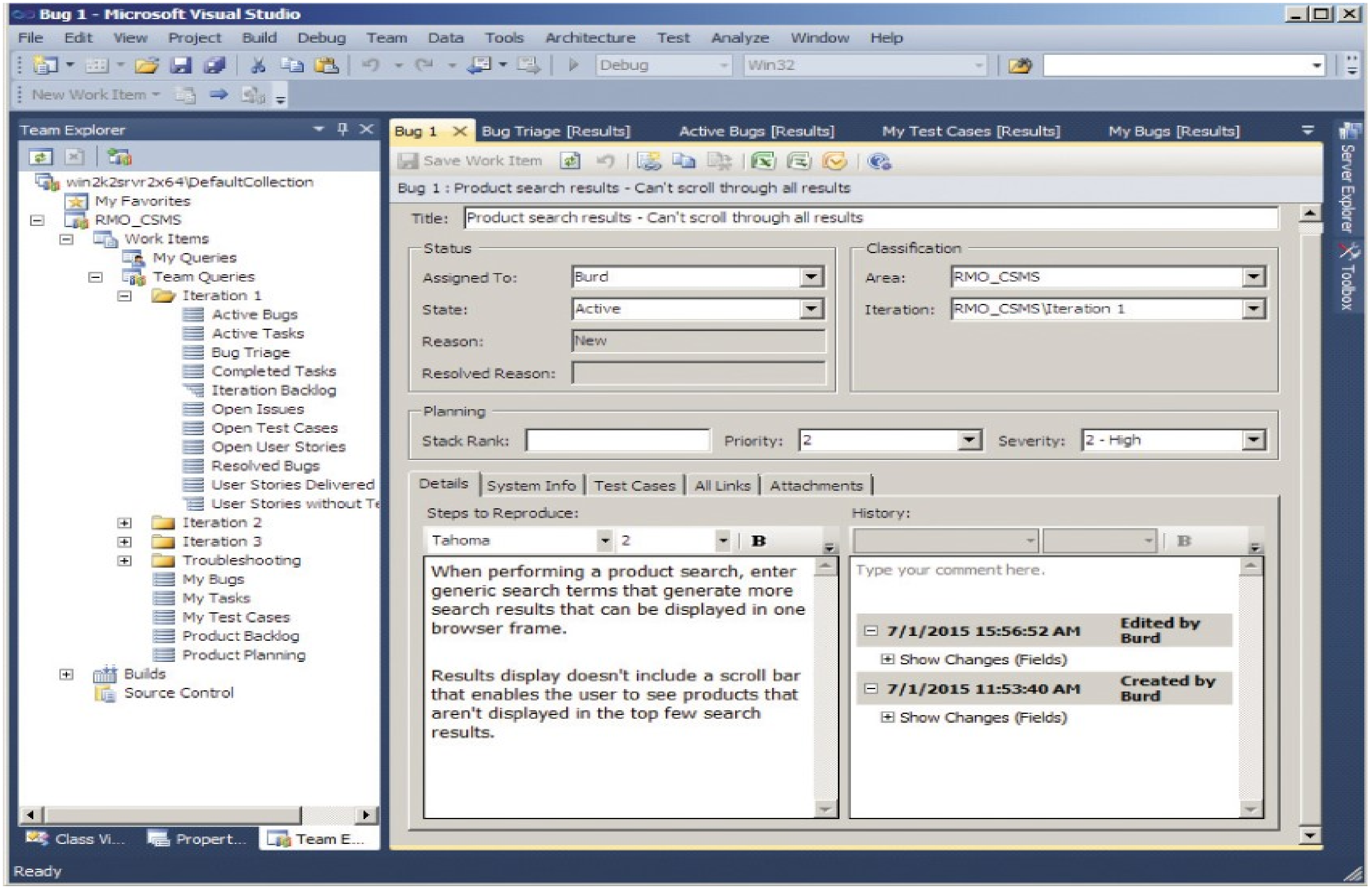
Name:	Windows 8.1 (build 9600), 64-bit edition
Version:	6.3
Platform:	64 bit system

About box showing version number

Planning and Managing- Implementation, Testing and Deployment

- Submitting Error Reports and Change Requests
  - Standard reporting methods
  - Review of requests by a project manager or change control committee
  - For operational systems, extensive planning for design and implementation
- Implementing a Change
  - Identify what parts of the system must be changed
  - Secure resources (such as personnel) to implement the change
  - Schedule design and implementation activities
  - Develop test criteria and a testing plan for the changed system

Error Report



Bug 1 : Product search results - Can't scroll through all results

Title: Product search results - Can't scroll through all results

Status
Assigned To: Burd
State: Active
Reason: New
Resolved Reason:

Classification
Area: RMO_CSMS
Iteration: RMO_CSMS\Iteration 1

Planning
Stack Rank:
Priority: 2
Severity: 2 - High

Details | System Info | Test Cases | All Links | Attachments

Steps to Reproduce:

Tahoma 2 B

When performing a product search, enter generic search terms that generate more search results that can be displayed in one browser frame.

Results display doesn't include a scroll bar that enables the user to see products that aren't displayed in the top few search results.

History:

Type your comment here.

7/1/2015 15:56:52 AM Edited by Burd
Show Changes (Fields)

7/1/2015 11:53:40 AM Created by Burd
Show Changes (Fields)

Error report in Visual Studio

Summary

- Three layer design is an architectural design pattern, part of the movement toward the use of design principles and patterns.
- Implementation and deployment are complex processes because they consist of so many interdependent activities
- Implementation activities include program the software, unit tests, building test cases, and integrate and test components
- Deployment activities include perform system and stress tests, perform acceptance tests, convert existing data, build training materials/conduct training, configure and set up the production environment, and deploy the solution
- Testing is a key activity of implementation and deployment and includes unit tests, integration tests, usability tests, system/performance/stress tests, and acceptance tests
- A program development plan is a trade-off among available resources, available time, and the desire to detect and correct errors prior to system deployment
- Configuration and change management activities track changes to models and software through multiple system versions, which enables developers to test and deploy a system in stages
- Versioning also improves post deployment support by enabling developers to track problem support to specific system versions
- Source code control systems enable development teams to coordinate their work
- Three options for deployment include direct deployment, parallel deployment and phased deployment
- Direct deployment is riskier but less expensive. Parallel deployment is less risky but more expensive
- For moderate to large projects, a phase deployment approach makes sense to get key parts of the system operational earlier