# Advanced Software Engineering Techniques

Course 9 – 19 December 2023

Adrian Iftene
adiftene@info.uaic.ro

# Content

- Recapitulation
  - QoS
  - Functional Testing
  - Non-Functional Testing
- Rotting Design
- Refactoring
- Deployment, maintenance, release

# R – Quality of service

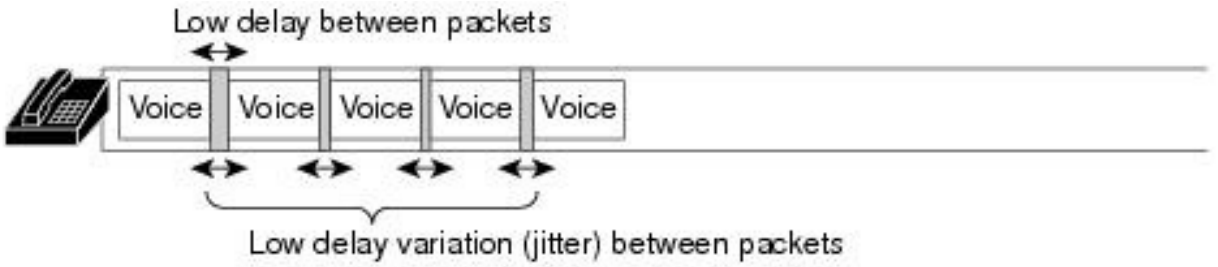
- QoS = ability to provide **different priority** to different **applications**, **users**, or **data flows**, or to **guarantee a certain level of performance** to a data flow

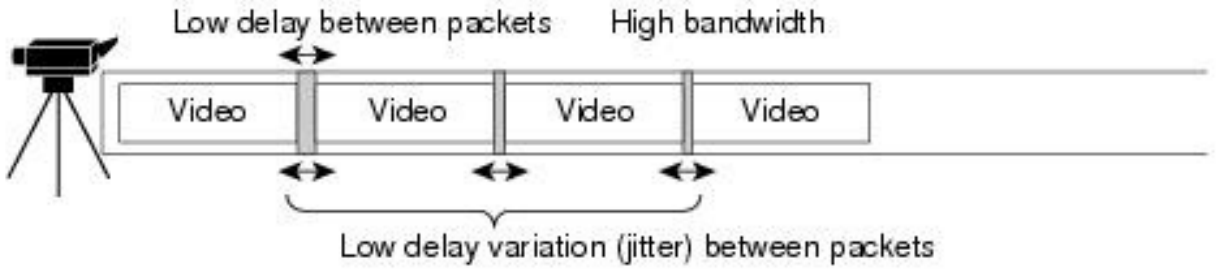
- Where? Computer networking, telecommunication networks

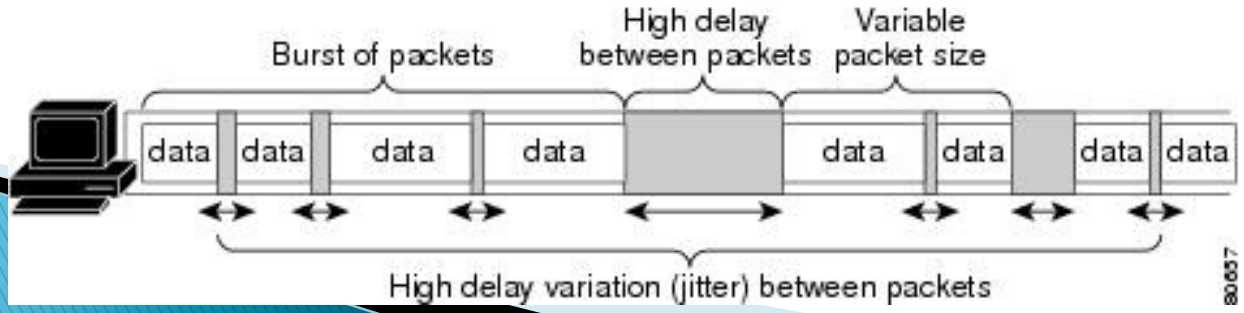
- How? A network may agree on a **traffic contract** with the application software. The network must guarantee a required **bit rate**, **delay**, **jitter**, **packet dropping** probability and/or **bit error rate**

# R – QoS – Example

- Voice, Video, and Data Transmission Requirements



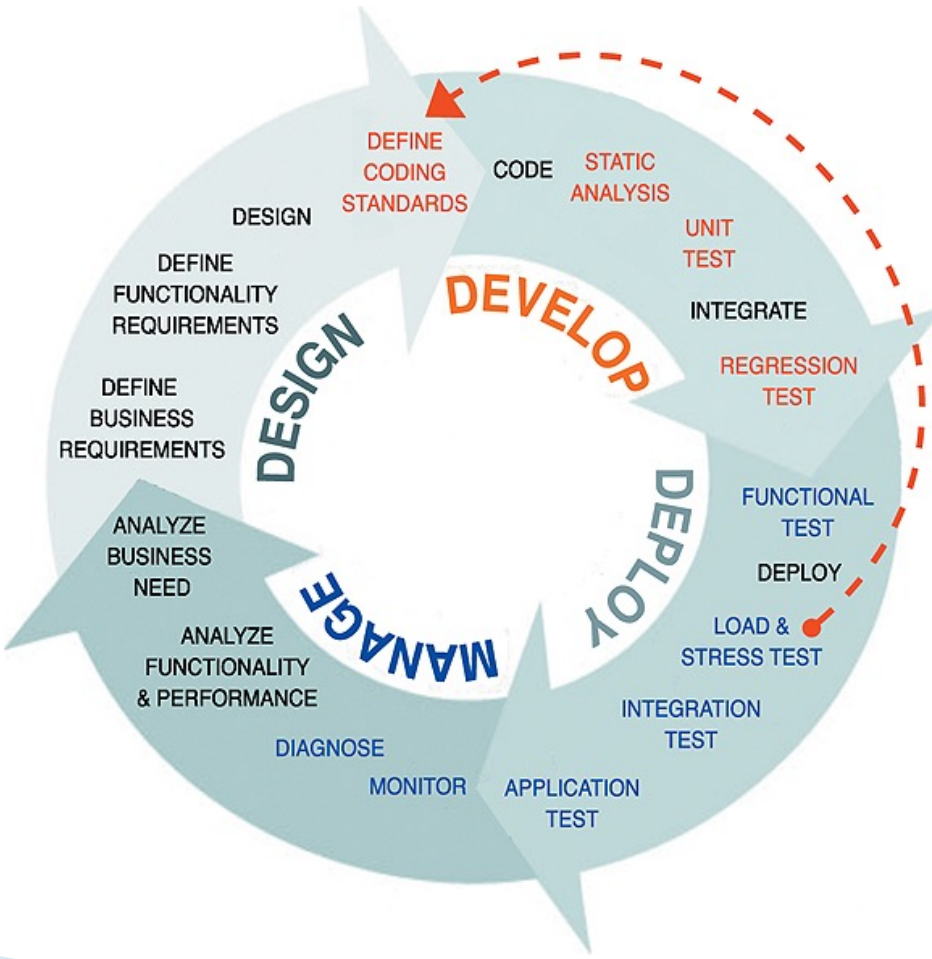



# R – QoS – How to obtain it?

- **In advance**: by prioritizing traffic
- **Reserving resources**: Resources are reserved at each step on the network for the call as it is set up
- **Over provisioning**: a network capacity is based on peak traffic load estimates
- **Integrated services**: reserving network resources

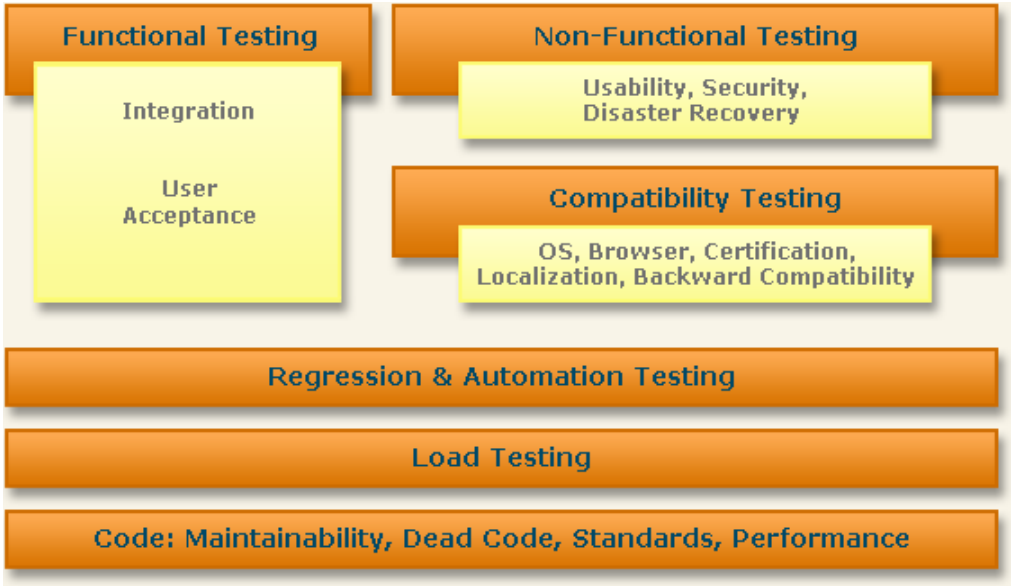
# R – Functional Testing

- Testing conducted on a complete, integrated system to evaluate the system's compliance with its **specified requirements**
- Unit Testing
- Integration Testing
- Ad-Hoc Testing
- Regression Testing
- User Acceptance Testing
- Installation Testing
- Interface & Usability Testing
- System Testing
- White Box & Black Box Testing



# R – Non-Functional Testing

- Load and Performance
- Ergonomics
- Stress & Volume
- Compatibility & Migration
- Data Conversion
- Security / Penetration
- Operational Readiness
- Installation
- Security Testing (Application Security, Network Security, System Security)



# Content

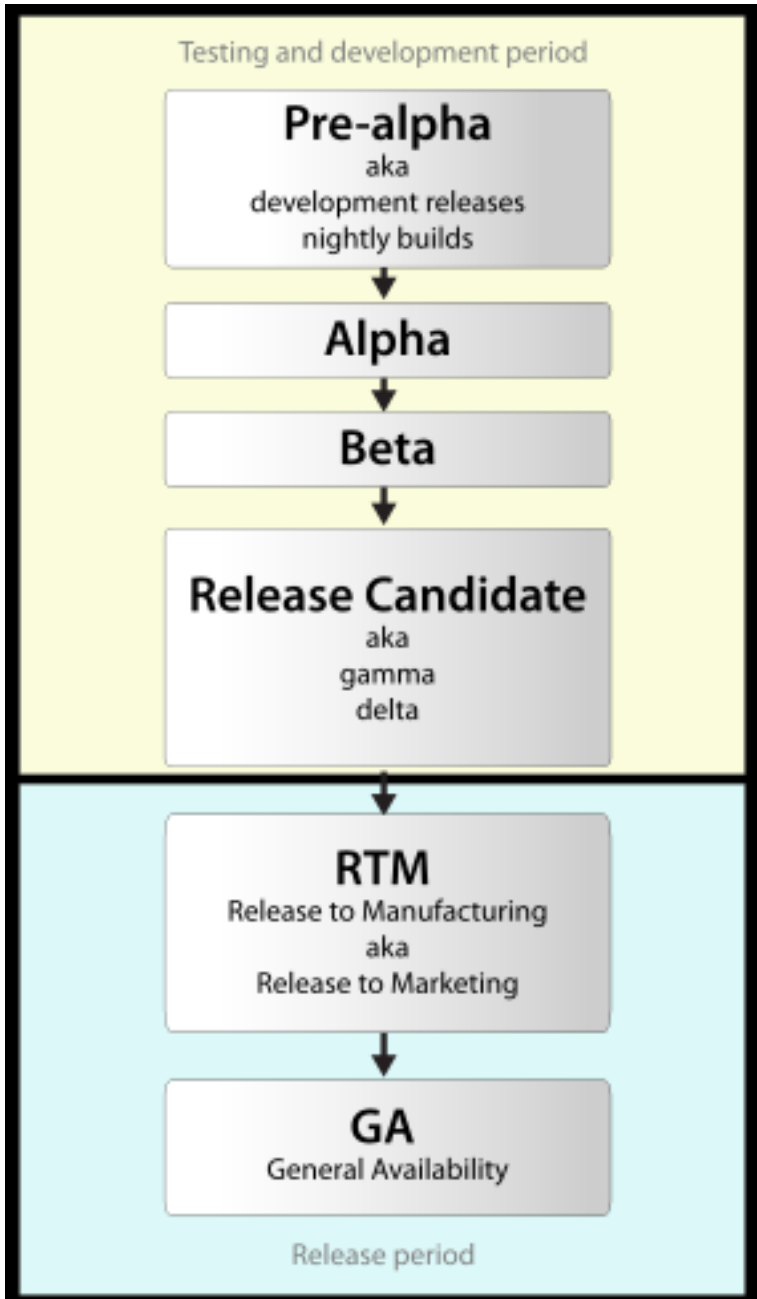
- Recapitulation
  - QoS
  - Functional Testing
  - Non-Functional Testing
- **Rotting Design** 
- Refactoring
- Deployment, maintenance, release

# Software Architecture

- Design Principles and Design Patterns. Robert C. Martin. www.objectmentor.com
- *What is software architecture*? The answer is **multitiered**. This is the domain of design patterns, packages, components, and classes
- **Highest Level**: architecture patterns and structure of software applications
- **Middle Level**: architecture that is specifically related to the purpose of the software application
- **Lowest Level**: the architecture of the modules and their interconnections

# Software Life Cycle

- **First Release**: clean, elegant, and compelling
- But then something begins to happen => The software starts to rot => **in time** => The program becomes a festering mass of code that the developers find increasingly hard to maintain
- What about a new change?
- How to redesigned project?



# Symptoms of Rotting Design

- There are four primary symptoms:
  - **Rigidity** – the tendency for software to be difficult to change, even in simple ways
  - **Fragility** – the tendency of the software to break in many places every time it is changed
  - **Immobility** – the inability to reuse software from other projects or from parts of the same project
  - **Viscosity** – it is easy to do the wrong thing, but hard to do the right thing

# Rotting Design – Rigidity

- **Every change causes a cascade of subsequent changes in dependent modules** (instead a simple two day change we have a multi-week marathon of change in module after module)
- When software behaves this way, **managers fear to allow engineers to fix non-critical problems**
- When the manager's fears become so acute that they refuse to allow changes to software, official rigidity sets in. Thus, what starts as a design deficiency, winds up being adverse management policy

# Rotting Design – Fragility

- **Every fix makes it worse**, introducing more problems than are solved. Such software is impossible to maintain
- **Managers**: Every time they authorize a fix, they fear that the software will break in some unexpected way
- Such software causes managers and customers to suspect that the **developers have lost control of their software**

# Rotting Design – Immobility

- What happens? **Software is simply rewritten instead of reused**
- It often happens that one engineer will discover that he needs a module that is similar to one that another engineer wrote. However, it also often happens that **the module in question has too much baggage that it depends upon**
- After much work, the engineers discover that the work and risk required to separate the desirable parts of the software from the undesirable parts are too great to tolerate

# Rotting Design – Viscosity

- Comes in two forms: viscosity of the design, and viscosity of the environment
- **Viscosity of the design**: When faced with a change, engineers usually find more than one way to make the change. Some of the ways preserve the design, others do not
- **Viscosity of environment** comes about when the development environment is slow and inefficient. For example, if compile times are very long, engineers will be tempted to make changes that don't force large recompiles, even though those changes are not optimal from a design point of view

# Rotting Design – Causes

- **Changing Requirements** – The requirements have been changing in ways that the initial design did not anticipate
  - If our designs are failing due to the constant rain of changing requirements, it is our designs that are at fault
- **Dependency Management** – What kind of changes cause designs to rot? Changes that introduce new and unplanned for dependencies
  - In order to forestall the degradation of the dependency architecture, the dependencies between modules in an application must be managed (by dependency firewalls)

# Content

- Recapitulation
  - QoS
  - Functional Testing
  - Non-Functional Testing
- Rotting Design
- **Refactoring** 
- Deployment, maintenance, release

# Refactoring

- "*Refactoring is the process of changing a software system in such a way that it **does not alter the external behavior** of the code yet improves its internal structure*." – Martin Fowler in *Refactoring Improving The Design Of Existing Code*

- Practically, refactoring means **making code clearer and cleaner and simpler and elegant**

- Refactoring implies **equivalence**; **the beginning and end products must be functionally identical**

# Refactoring – Maths

- Refactoring is a kind of **reorganization.** Technically, it comes from **mathematics** when **you factor an expression into an equivalence** – the factors are cleaner ways of expressing the same statement
- Example: Count the terms and operators in

$$(x - 1) * (x + 1) = x^2 - 1$$

- Four terms versus three. Three operators versus two.
- The left side expression is simpler to understand because it uses simpler operations. Also, it provides the roots of function $f(x) = x^2 - 1$

# Refactoring – Motivation

- Refactoring is a good thing because complex expressions are typically built from **simpler**, more grokkable components
- Code refactoring is the process of changing a computer program's source code without modifying its external functional behavior **in order to improve** some of the **nonfunctional attributes** of the software
- Advantages include improved **code readability** and **reduced complexity to improve the maintainability** of the source code, as well as a **more expressive internal architecture or object model to improve extensibility**

# Refactoring – Attention!

- Refactoring doesn't prevent **changing functionality**, it just says that it's a different activity from rearranging code
- The key insight is that it's easier to rearrange the code correctly **if you don't simultaneously try to change its functionality**
- Refactoring is **not** rewriting. The difference between the two is that refactoring doesn't change the functionality of the system whereas rewriting does

# How to do Refactoring?

- Refactoring is done in **small steps**
- Each transformation (called a "**refactoring**") does little, but a sequence of transformations can produce a significant restructuring
- Examples would run the range from **renaming a variable** to **introducing a method** into a third-party class that you don't have source for
- Since each refactoring is small, it's less likely to go wrong. The system is also kept fully working after each small refactoring, reducing the chances that a system can get seriously broken during the restructuring

# Refactoring – Benefits

- **Immediate results**: Decreased coupling and Increased cohesion
- **Maintainability**: It is easier to fix bugs because the source code is easy to read and the intent of its author is easy to grasp
- **Extensibility**: It is easier to extend the capabilities of the application if it uses recognizable design patterns, and it provides some flexibility where none before may have existed

# Refactoring – Signals

- Duplicate Code
- Long Methods
- Large classes
- Long lists of parameters
- Instructions **switch**
- Speculative generality
- Intense communication between objects
- Chaining Message

# Refactoring – Steps

- **Before**: A solid set of automatic unit tests is needed. The tests should demonstrate in a few seconds that the behavior of the module is correct

- **During the process**: The process is an iterative cycle of
  - Making a small program transformation
  - Testing it to ensure correctness
  - If at any point a test fails, you undo your last small change and try again in a different way
- Through many small steps the program moves from where it was to where you want it to be

# Refactoring – Examples and Solutions

- Examples:
  - reducing large monolithic routines into a set of individually concise, well-named, single-purpose methods
  - moving a method to a more appropriate class
  - removing misleading comments
- Possible Solutions:
  - For a long routine, extract one or more smaller subroutines
  - For duplicate routines, remove the duplication and utilize one shared function in their place

# Refactoring techniques (1)

- Techniques that allow for more abstraction
  - **Encapsulate Field** – force code to access the field with getter and setter methods
  - **Generalize Type** – create more general types to allow for more code sharing
  - **Replace type** – checking code with State/Strategy
  - Replace conditional with polymorphism
- Techniques for breaking code apart into more logical pieces
  - **Extract Method**, to turn part of a larger method into a new method. By breaking down code in smaller pieces, it is more easily understandable. This is also applicable to functions.
  - **Extract Class** moves part of the code from an existing class into a new class.

# Refactoring techniques (2)

- Techniques for improving names and location of code
  - **Move Method** or **Move Field** – move to a more appropriate Class or source file
  - **Rename Method** or **Rename Field** – changing the name into a new one that better reveals its purpose
  - **Pull Up** – in OOP, move to a superclass
  - **Push Down** – in OOP, move to a subclass

# Automated code refactoring (1)

- Many software editors and IDEs have automated refactoring support
- IntelliJ IDEA (for Java)
- Eclipse's Java Development Toolkit (JDT)
- NetBeans (for Java)
- Embarcadero Delphi
- Visual Studio (for .NET)
- JustCode (addon for Visual Studio)
- ReSharper (addon for Visual Studio)
- Coderush (addon for Visual Studio)

# Automated code refactoring (2)

- Visual Assist (addon for Visual Studio with refactoring support for VB, VB.NET. C# and C++)
- DMS Software Reengineering Toolkit (Implements large-scale refactoring for C, C++, C#, COBOL, Java, PHP and other languages)
- Photran a Fortran plugin for the Eclipse IDE
- SharpSort addin for Visual Studio 2008
- Sigasi HDT (for VHDL)
- Xcode
- Smalltalk Refactoring Browser (for Smalltalk)
- Simplifide (for Verilog, VHDL and SystemVerilog)

# Refactoring – *Where to start?*

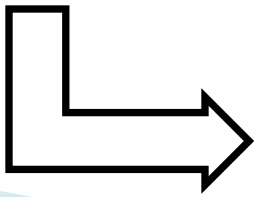
- http://sourcemaking.com/refactoring
- **Composing Methods** – methods to package code properly
- **Moving Features Between Objects** – deciding where to put responsibilities
- **Organizing Data** – make working with data easier
- **Simplifying Conditional Expressions**
- **Making Method Calls Simpler** – make interfaces more straightforward
- **Dealing with Generalization** – moving methods around a hierarchy of inheritance
- **Big Refactoring** – a sense of the whole "game"

# Refactoring – Remove Assignments to Parameters

- The reason: lack of clarity and to confusion between *pass by value* and *pass by reference*
- **Mechanics**
  - Create a temporary variable for the parameter.
  - Replace all references to the parameter, made after the assignment, to the temporary variable.
  - Change the assignment to assign to the temporary variable.
  - Compile and test.

```
int discount (int inputVal, int quantity, int yearToDate) {
  if (inputVal > 50) inputVal -= 2;
  if (quantity > 100) inputVal -= 1;
  if (yearToDate > 10000) inputVal -= 4;
  return inputVal;
}
```

Wrong



Correct

```
int discount (int inputVal, int quantity, int yearToDate) {
  int result = inputVal;
  if (inputVal > 50) result -= 2;
  if (quantity > 100) result -= 1;
  if (yearToDate > 10000)  result -= 4;
  return result;
}
```

# Refactoring – Introduce Explaining Variable

- **When you have a complicated expression**: Put the result of the expression, or parts of the expression, in a temporary variable with a name that explains the purpose. **Mechanics:**
  - Declare a final temporary variable, and set it to the result of part of the complex expression.
  - Replace the result part of the expression with the value of the temp.
  - If the result part of the expression is repeated, you can replace the repeats one at a time
  - Compile and test.
  - Repeat for other parts of the expression.

Wrong

```
if ((platform.toUpperCase().indexOf("MAC") > -1)
    && (browser.toUpperCase().indexOf("IE") > -1)
    && wasInitialized() && resize > 0 ) {
      // do something
}
```

Correct

```
final boolean isMacOs = platform.toUpperCase().indexOf("MAC") > -1;
final boolean isIEBrowser = browser.toUpperCase().indexOf("IE") > -1;
final boolean wasResized = resize > 0;
if (isMacOs && isIEBrowser && wasInitialized() && wasResized) {
    // do something
}
```

# Refactoring – Replace Array with Object (1)

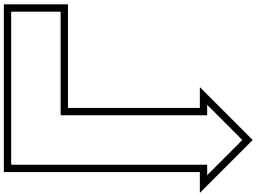
- When you have an array in which certain elements mean different things. **Mechanics:**
- **Step 1**
  - Create a new class to represent the information in the array. Give it a public field for the array.
  - Change all users of the array to use the new class.
  - Compile and test.
- Step 2
  - One by one, add getters and setters for each element of the array. Name the assessors after the purpose of the array element. Change the clients to use the assessors. Compile and test after each change.
  - When all array accesses are replaced by methods, make the array private.
  - Compile.
- Step 3:
  - For each element of the array, create a field in the class and change the assessors to use the field.
    Compile and test after each element is changed.
    When all elements have been replaced with fields, delete the array.

# Refactoring – Replace Array with Object (2)

- **Solution**: Replace the array with an object that has a field for each element

Wrong

```
String[] row = new String[3];
row [0] = "Liverpool";
row [1] = "15";
```



```
Performance row = new Performance();
row.setName("Liverpool");
row.setWins("15");
```

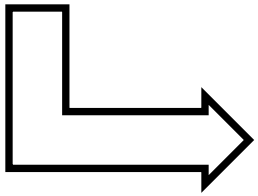
Correct

# Refactoring – Consolidate Duplicate Conditional Fragments

- When the same fragment of code is in all branches of a conditional expression
- Solution: *Move it outside of the expression*

Wrong

```
if (isSpecialDeal()) {
  total = price * 0.95;
  send();
}
else {
  total = price * 0.98;
  send();
}
```



```
if (isSpecialDeal())
  total = price * 0.95;
else
  total = price * 0.98;

send();
```

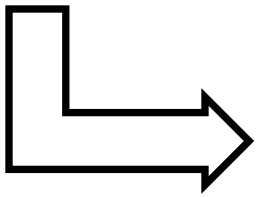
Correct

# Refactoring – Preserve Whole Object

- When you are getting several values from an object and passing these values as parameters in a method call.
- **Solution**: *Send the whole object instead*

Wrong

```
int low = daysTempRange().getLow();
int high = daysTempRange().getHigh();
withinPlan = plan.withinRange(low, high);
```

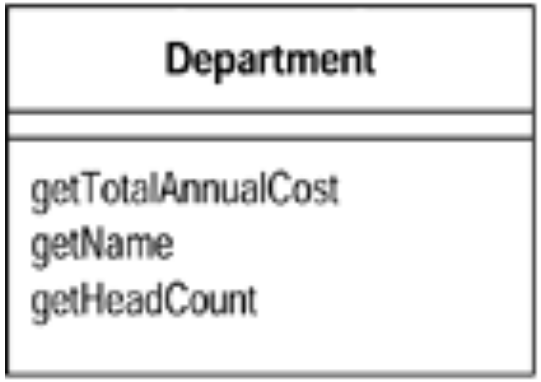


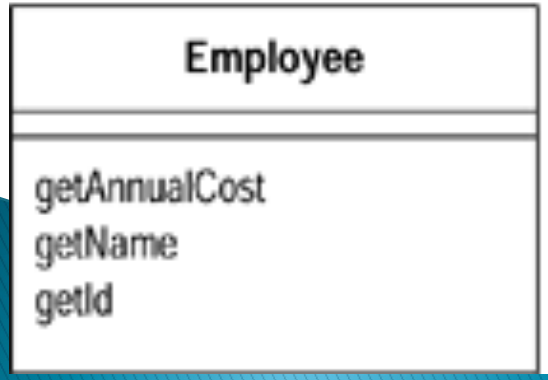
```
withinPlan = plan.withinRange(daysTempRange());
```

Correct

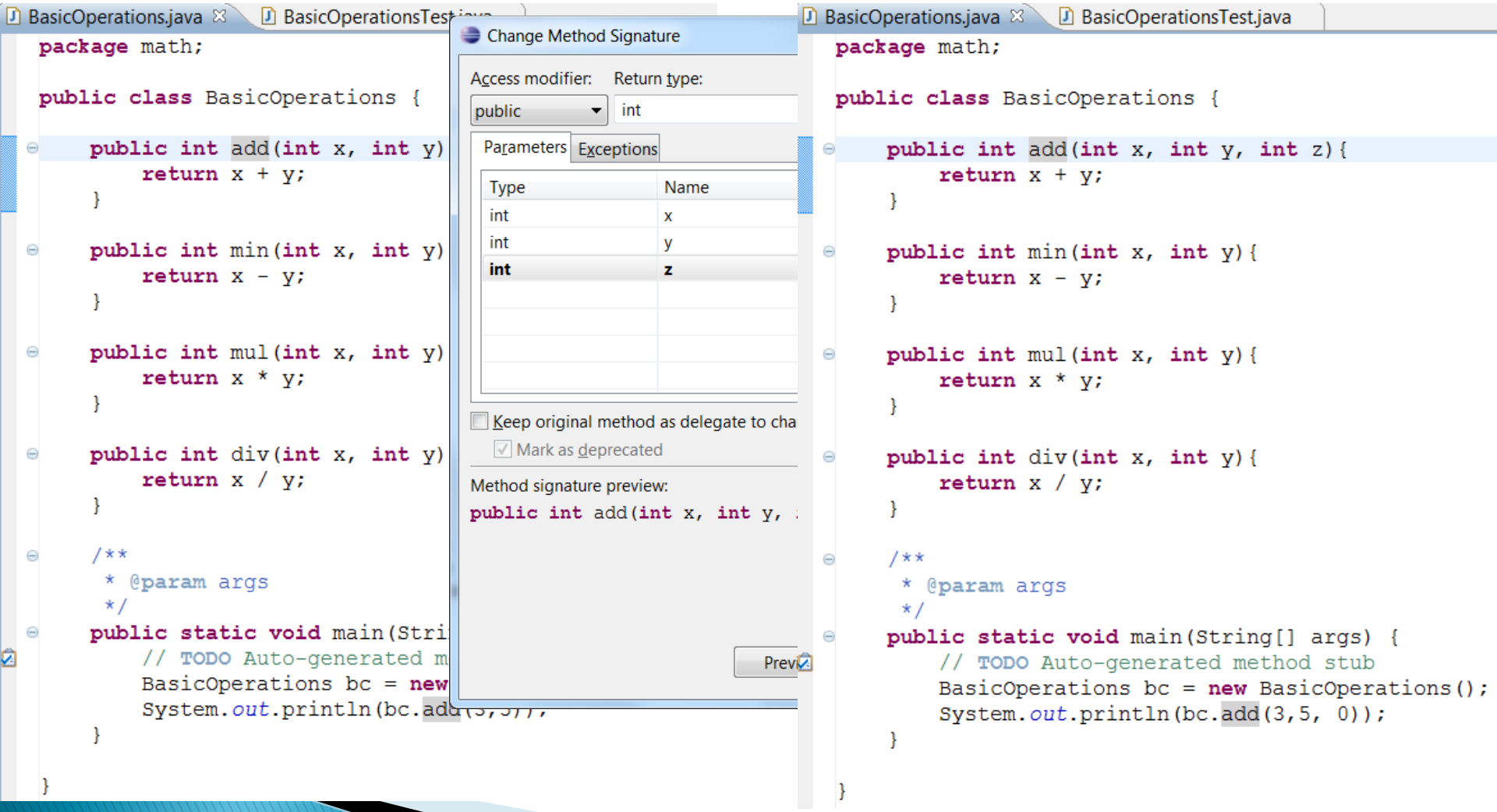
# Refactoring – Extract Superclass

- When you have two classes with similar features.
- Solution: *Create a superclass and move the common features to the superclass.*



# Refactoring – Change method signature



# Content

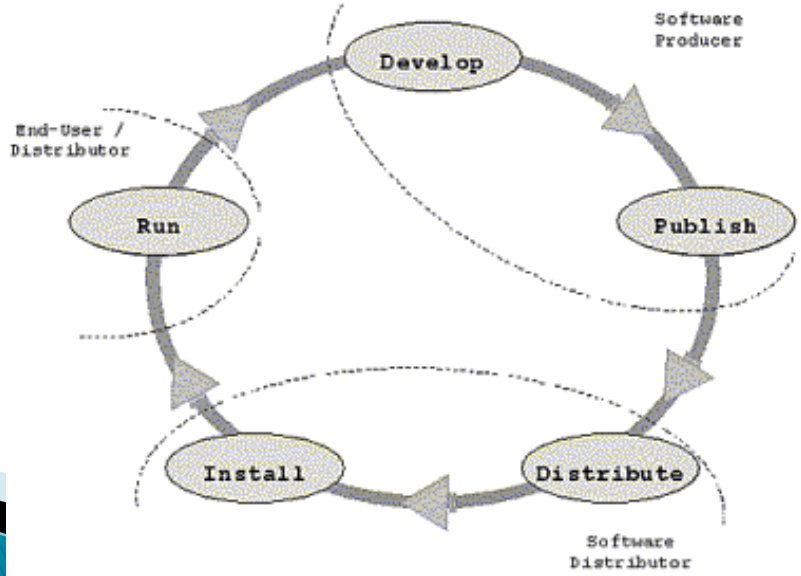
- Recapitulation
  - QoS
  - Functional Testing
  - Non-Functional Testing
- Rotting Design
- Refactoring
- **Deployment, maintenance, release** 

# Deployment and maintenance

- **Deployment** starts after the code is appropriately tested, is approved for release and sold
- **Software Training and Support** is very important to have training classes for new clients of your software
- **Maintaining** and enhancing software to cope with newly discovered problems or new requirements can take far more time than the initial development of the software
- **Bug Tracking System tools** are often deployed at this stage of the process to allow development teams to interface with customer/field teams testing the software to identify any real or perceived issues

# Software Deployment

- All of the activities that make a software system available for use
- "Deployment" should be interpreted as a *general process* that has to be customized according to specific requirements or characteristics
- At this step code must be distributed into a production environment



# Deployment Activities (1)

- **Release** – includes all the operations to prepare a system for assembly and transfer to the customer site
- **Installation** – on a production server in a production environment or in a test environment
- **Activation** – the activity of starting up the executable component of software
- **Deactivate** – inverse of activation – refers to shutting down any executing components of a system
- **Adapt** – is a process to modify a software system that has been previously installed (local events such as changing the environment of customer site)
- **Update** – The update process replaces an earlier version of all or part of a software system with a newer release

# Deployment Activities (2)

- **Built-In** – Mechanisms for installing updates are built into some software systems
- **Version tracking** – Version tracking systems help the user find and install updates to software systems installed on PCs and local networks (Web based version, Local version, Browser based version)
- **Uninstall** – the inverse of installation. It is the removal of a system that is no longer required
- **Retire** – Ultimately, a software system is marked as obsolete and support by the producers is withdrawn. It is the end of the life cycle of a software product

# Software Maintenance

- Software maintenance in software engineering is **the modification of a software product after delivery** to correct faults, to improve performance or other attributes (Lehman 1969)
- Maintenance is **really evolutionary developments** and that maintenance decisions are aided by understanding what happens to systems (and software) **over time**
- A common perception of maintenance is that it is merely fixing bugs. However, studies and surveys over the years have indicated that the majority, over **80%, of the maintenance effort is used for non-corrective actions** (Pigosky 1997)

# Programs Types – Lehman and Belady

- **S-type** programs are those that *can be specified formally*
- **P-type** programs *cannot be specified*. Instead, an iterative process is used to find a working solution
- **E-type** programs are *embedded in the real world and become part of it*, thereby changing it. This leads to a feedback system where the program and its environment evolve in concert

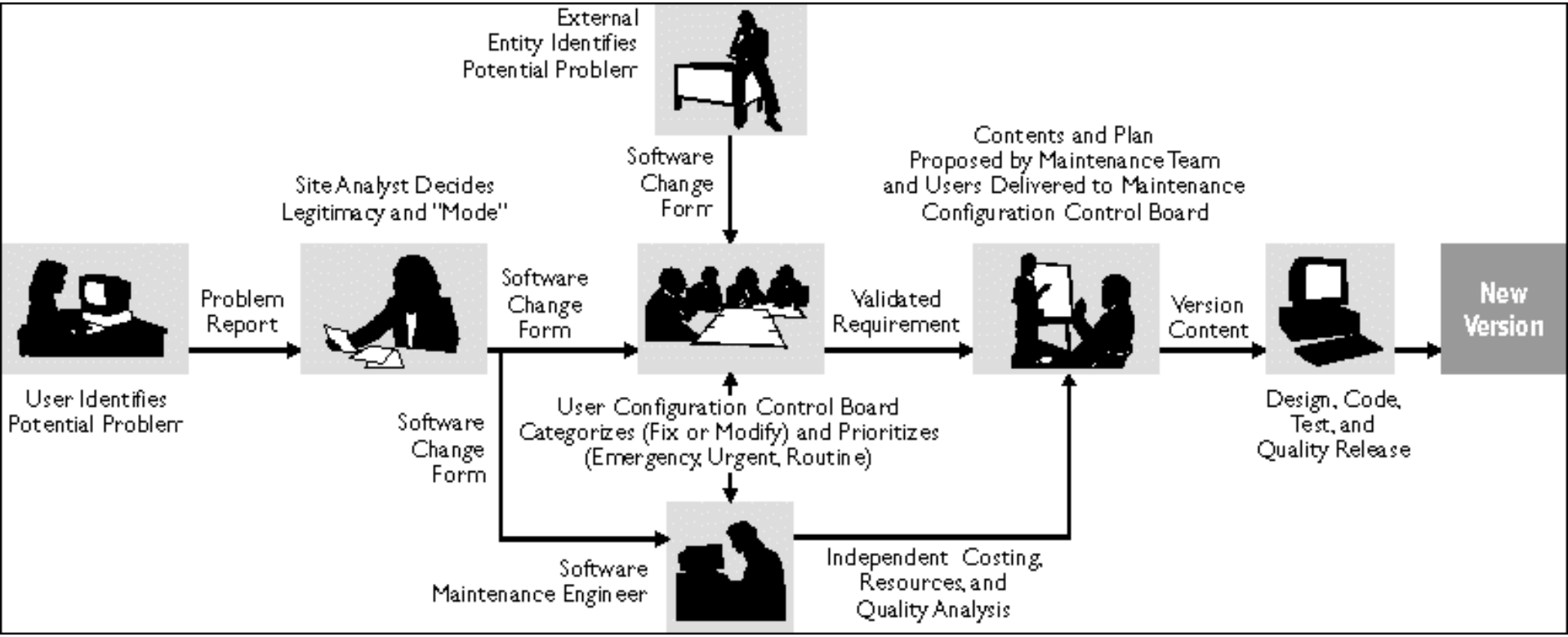
# Lehman's laws of software evolution (1)

1. (1974) **Continuing Change** — E-type systems must be continually adapted or they become progressively less satisfactory
2. (1974) **Increasing Complexity** — As an E-type system evolves its complexity increases unless work is done to maintain or reduce it
3. (1974) **Self Regulation** — E-type system evolution process is self regulating with distribution of product and process measures close to normal
4. (1978) **Conservation of Organizational Stability** (invariant work rate) - The average effective global activity rate in an evolving E-type system is invariant over product lifetime

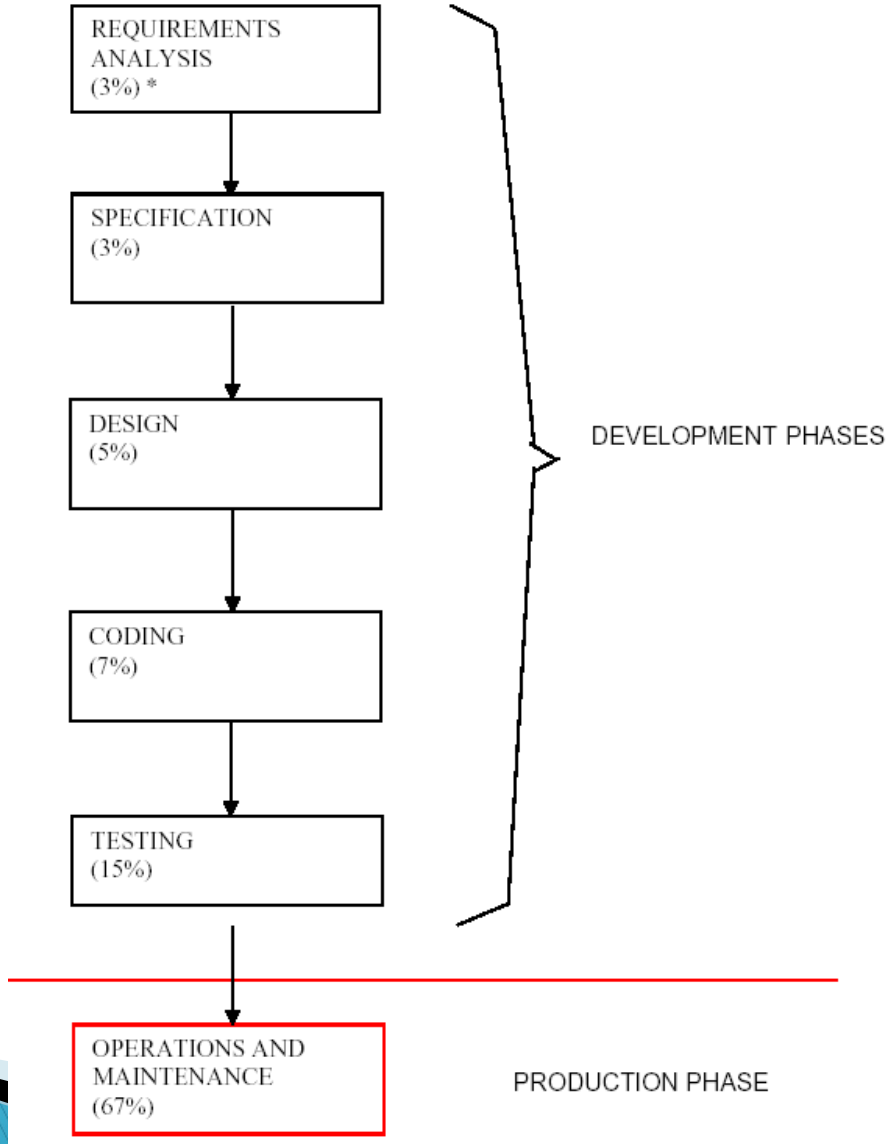
# Lehman's laws of software evolution (2)

5. (1978) **Conservation of Familiarity** — As an E-type system evolves all associated with it, developers, sales personnel, users. Excessive growth diminishes that mastery. Hence the average incremental growth remains invariant as the system evolves
6. (1991) **Continuing Growth** — The functional content of E-type systems must be continually increased to maintain user satisfaction over their lifetime
7. (1996) **Declining Quality** — The quality of E-type systems will appear to be declining unless they are rigorously maintained and adapted to operational environment changes
8. (1996) **Feedback System** (first stated 1974, formalized as law 1996) — E-type evolution processes constitute multi-level, multi-loop, multi-agent feedback systems and must be treated as such to achieve significant improvement over any reasonable base

# Software Maintenance Implications on Cost and Schedule

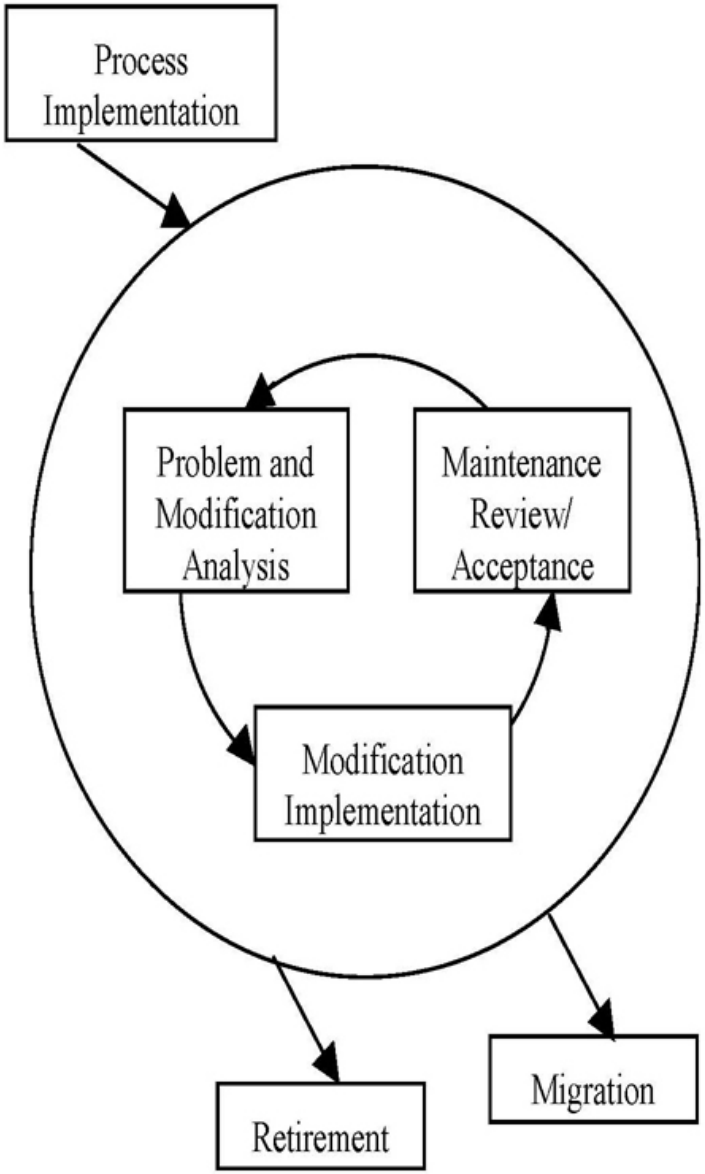


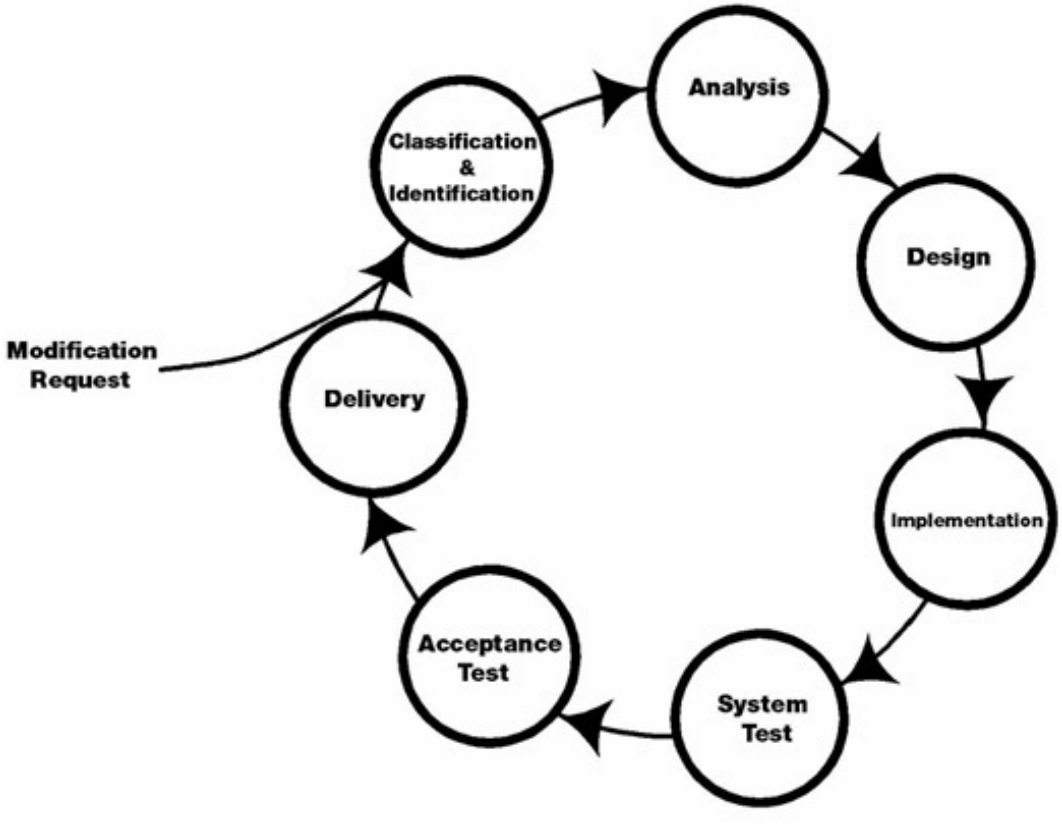
# Software Development Costs



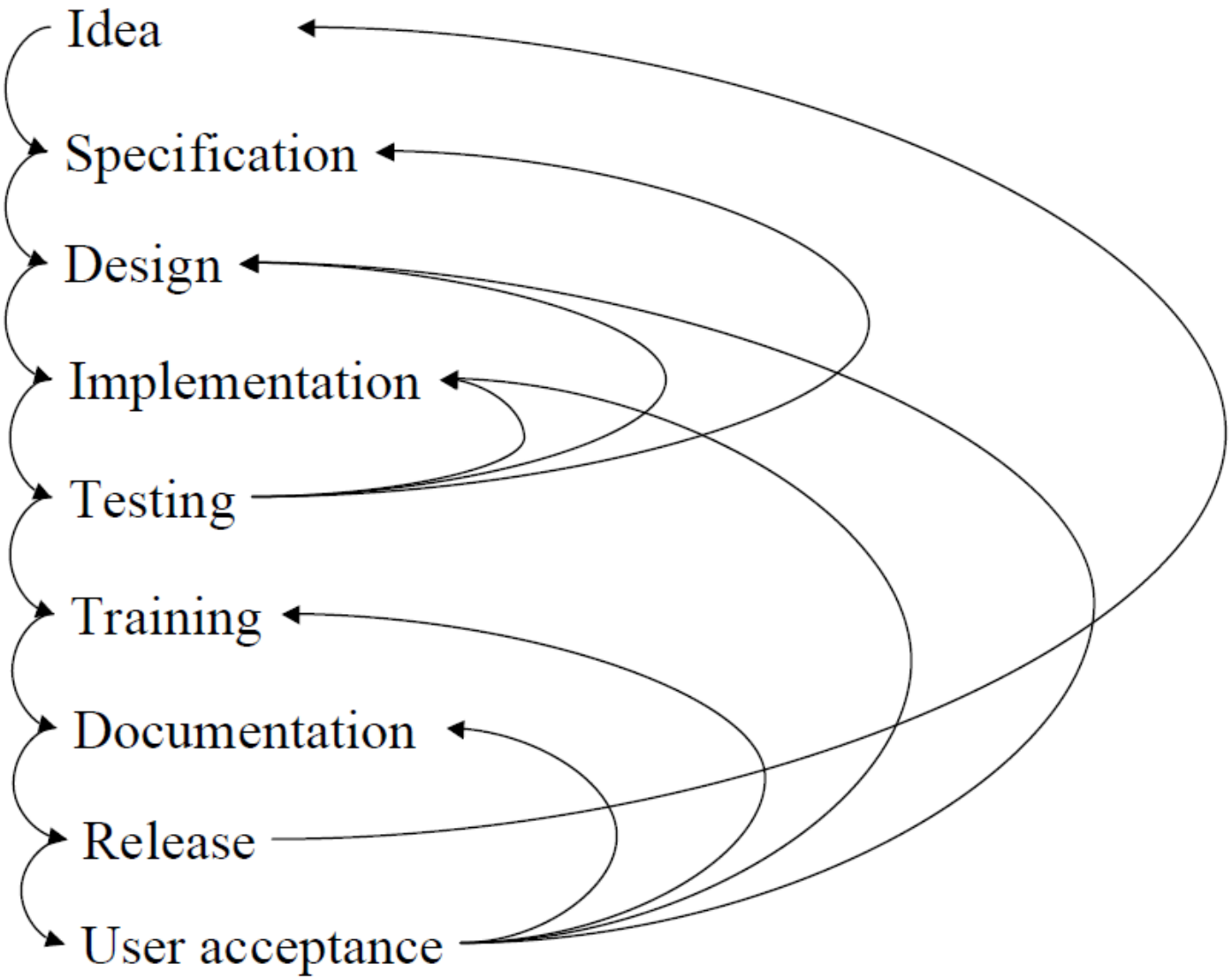

* The percentages above indicate relative costs.

# Software Maintenance Processes



# Real Maintenance/Development Environment

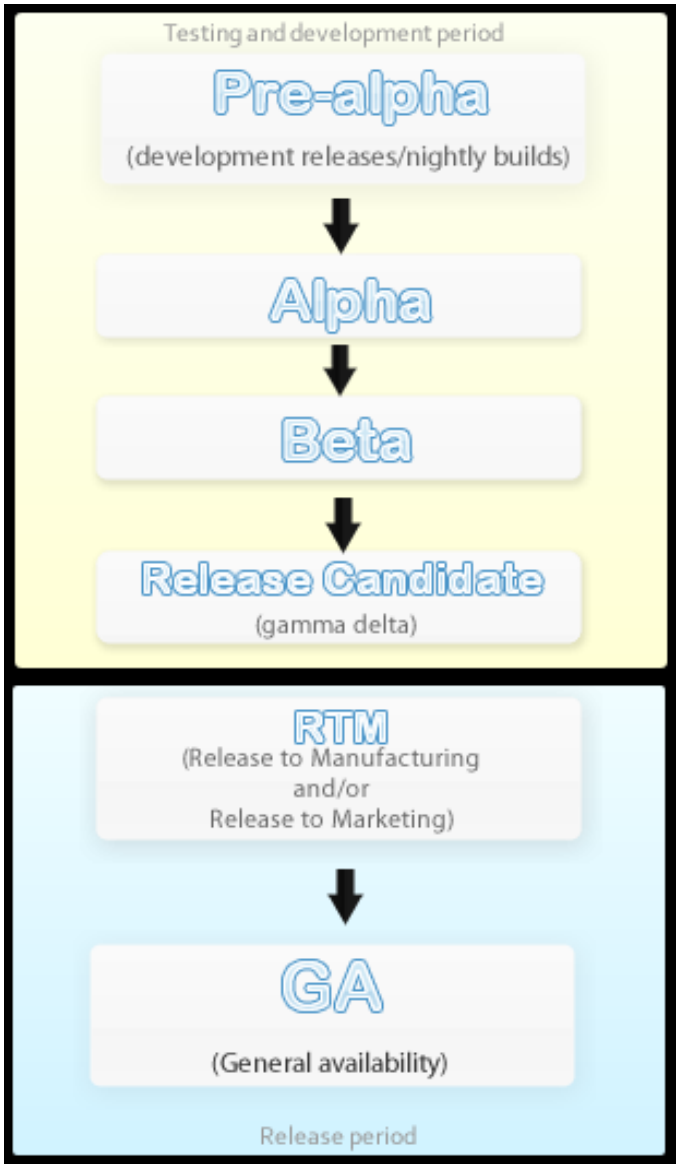


# Categories of maintenance in ISO/IEC 14764

- **Corrective maintenance**: Reactive modification of a software product performed after delivery to *correct discovered problems*
- **Adaptive maintenance**: Modification of a software product performed after delivery to *keep a software product usable in a changed or changing environment*
- **Perfective maintenance**: Modification of a software product after delivery to *improve performance or maintainability*
- **Preventive maintenance**: Modification of a software product after delivery to *detect and correct latent faults in the software product before they become effective faults*

# Software release life cycle

- A software release is the **distribution of software code, documentation, and support materials**
- The software release life cycle is composed of discrete phases from **planning and development** to **release** and **support** phases
- Software Development and Release Stages:
  - Testing and Development period
  - Release period
  - Support period



# Software Development Stages

- **Pre-alpha** – refers to all activities performed during the software project prior to testing
- **Alpha** – is the first phase to begin software testing
- **Beta** – The focus of beta testing is reducing impacts to users, often incorporating **usability testing.** *The process of delivering a beta version to the users is called* **beta release** *and this is typically the first time that the software is available outside of the organization that developed it*
- **Open and closed beta** – **closed beta** versions are released to a select group of individuals for a user test, while **open betas** are to a larger community group, sometimes to anybody interested
- **Release candidate** – refers to a version with potential to be a final product, ready to release unless fatal bugs emerge

# Release Stages

- RTM – The term "**release to manufacturing**" or "**release to marketing**" – is a term used when software is ready for or has been delivered or provided to the customer

- **General availability** or **general acceptance** (GA) is the point where all necessary commercialization activities have been completed and the software has been made available to the general market either via the web or physical media. It is also at this stage that the software is considered to have "**gone live**"

- A **live release** is considered to be very stable and relatively bug-free with a quality suitable for wide distribution and use by end users

# Support Stages

- **Service release** – During its supported lifetime, software is sometimes subjected to service releases, or service packs. Such service releases contain a collection of updates, fixes and/or enhancements, delivered in the form of a single installable package. They may also contain entirely new features

- **End-of-life** – When software is no longer sold or supported, the product is said to have reached end-of-life

# Bibliography


- Robert Cecil Martin: *Design Principles and Design Patterns.* 2000. http://www.objectmentor.com/resources/articles/Principles_and_Patterns.pdf
- Robert Cecil Martin: *Agile Development. Principles, Patterns, and Practices,* Prentice-Hall, 2003
- Refactoring: Improving the Design of Existing Code. Martin Fowler 1997. http://jaoo.dk/jaoo1999/schedule/MartinFowlerRefractoring.pdf

# Links

- What is Refactoring? http://c2.com/cgi/wiki?WhatIsRefactoring
- Refactoring Home Page: http://www.refactoring.com/
- Code Refactoring: http://en.wikipedia.org/wiki/Code_refactoring
- Refactoring. Where to start? http://sourcemaking.com/refactoring
- A Cyclic Model for Software Deployment: http://www.marinilli.com/publications/articles/2002/model.html
- Software Deployment: http://en.wikipedia.org/wiki/Software_deployment
- Software Maintenance: http://cnx.org/content/m14717/latest/
- Software Maintenance: http://en.wikipedia.org/wiki/Software_maintenance
- E. B. Swanson (Maintenance) http://www.anderson.ucla.edu/x1960.xml
- Software Maintenance Implications on Cost and Schedule: http://pro-software.biz/2009/06/software-maintenance-implications-on-cost-and-schedule/
- Maintenance: http://openseminar.org/se/modules/22/index/screen.do
- Software release life cycle: http://en.wikipedia.org/wiki/ Software_release_life_cycle
- Release engineering: http://en.wikipedia.org/wiki/Release_engineering
- Release management: http://en.wikipedia.org/wiki/Release_management