OOAD and SE

Software Design Principles

GRASP: Object Oriented Design Patterns

Dr. H.L. Phalachandra

Department of Computer Science and Engineering

leveraging most information from slides of

Prof. Vinay Joshi

Acknowledgements: Significant portions of the information in the slide sets presented through the course in the class, are extracted from the prescribed text books, information from the Internet and supplemented by my experience. Since these are only intended for presentation for teaching within PESU, there was no explicit permission solicited. We would like to sincerely thank and acknowledge that the credit/rights remain with the original authors/creators only

Design Principles

- If we need to build object oriented systems which are maintainable, extensible and modular, it would be best done using well known principles and practices as part of the OO design.

- These address problems or potential issues which developers will face and provide some principles or approaches to address the same or patterns which can be used to get over the same.

- These techniques have not been invented to create new ways of working, but to better document and standardize old, tried-and-tested programming principles in object-oriented design.

GRASP

stands for

General **R**esponsibility **A**ssignment **S**oftware **P**rinciples(or **P**atterns)

These are patterns and/or principles which help guide assigning responsibilities to collaborating classes objects.

These principles serve as guidelines to arrive at a better class design.

What is Responsibility

is a contract / obligation that a class / module / component must accomplish

- Responsibility can be:
  - accomplished by a single object
  - or a group of object collaboratively accomplish a responsibility.
- GRASP helps us in deciding which responsibility should be assigned to which object/class.
- Identify the objects and responsibilities from the problem domain, and also identify how objects interact with each other.
- Define blue print for those objects – i.e. class with methods implementing those responsibilities.

Responsibility

Design holds the responsibility of knowing the data & associated data and the actions or behaviours associated with data

- Knowing in terms of its
  - Private state
  - Computed state
- Expected Behaviour in terms
  - Send messages by itself and modifying its private state
    - Instantiate another object
    - Send messages to another object

GRASP Principles (9 of them)

1. Creator
2. Information Expert
3. Low Coupling
4. Controller
5. High Cohesion
6. Indirection
7. Polymorphism
8. Protected Variations
9. Pure Fabrication

1. Creator Principles

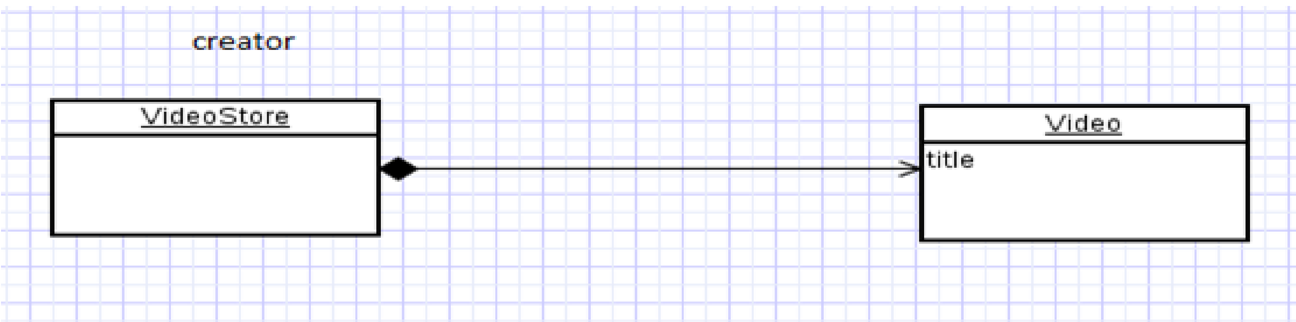
- Who creates an Object? Or who should create a new instance of some class?
- “Container” object creates “contained” objects.
- Decide who can be creator based on the objects association and their interaction.

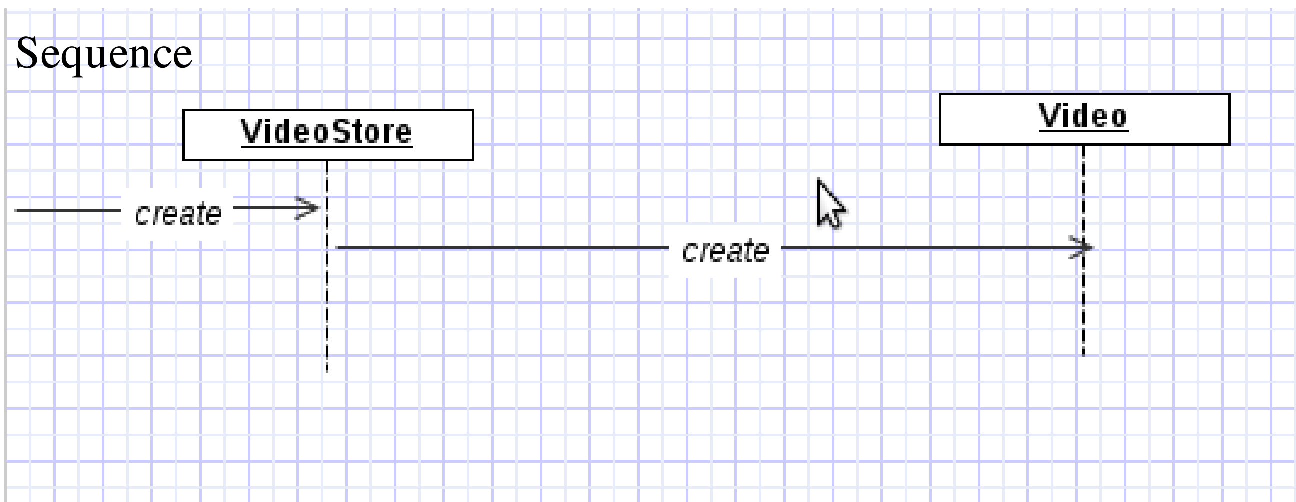
SOFTWARE DESIGN

Creator Example

- Consider VideoStore and Video in that store.
- VideoStore has an aggregation association with Video. i.e VideoStore is the container and the Video is the contained object.
- So, we can instantiate video object in a VideoStore class

Class Relation





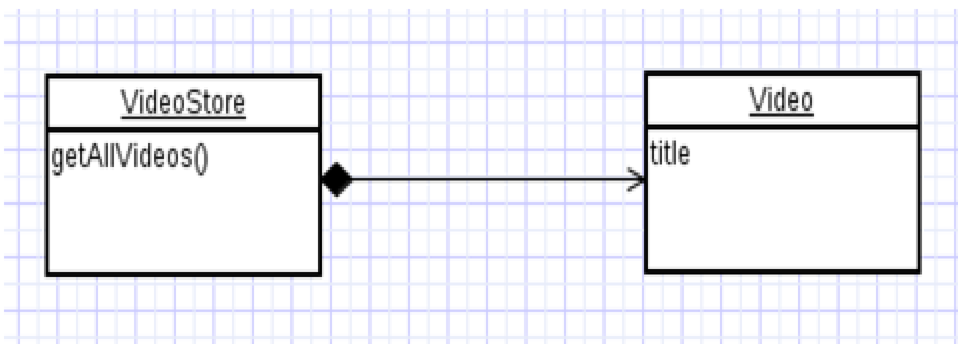
Information Expert

- Given an object O, which of the responsibilities can be assigned to O?
- Expert principle says – assign those responsibilities to a class which has the information to fulfill that responsibility.
- They have all the information needed to perform operations, or in some cases they collaborate with others to fulfill their responsibilities.

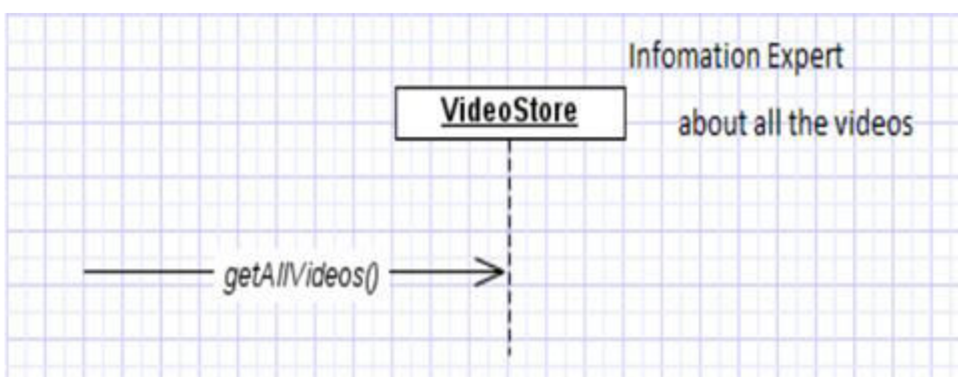
Information Expert Example

- Assume we need to get all the videos of a VideoStore.
- Since VideoStore knows about all the videos (catalog), we can assign this responsibility of giving all the videos can be assigned to VideoStore class.
- VideoStore is the information expert.

Class Relation



Sequence

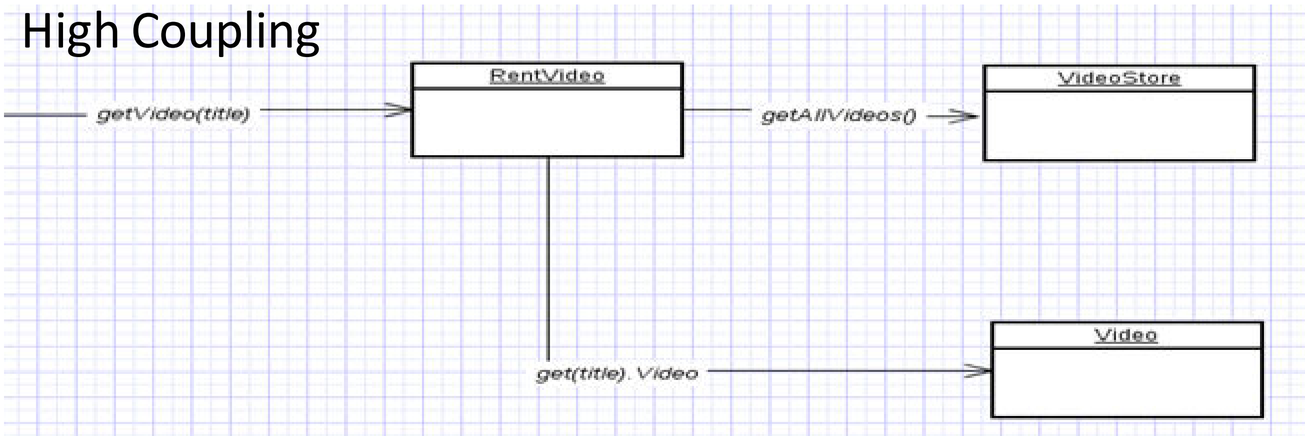


Low Coupling

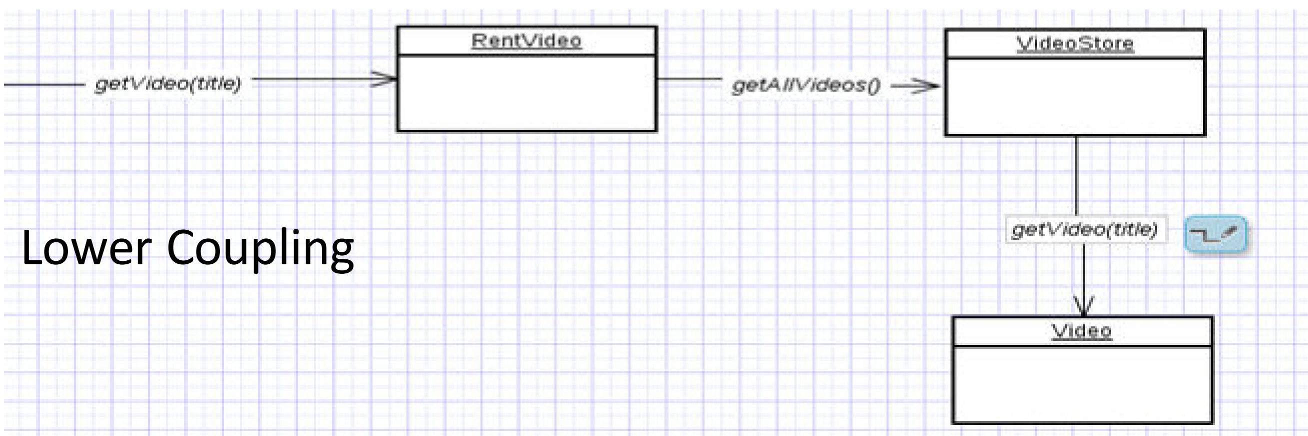
- How strongly the objects are connected to each other? Coupling – object depending on other object.
- When depended upon element changes, it affects the dependant also
- Low Coupling – How can we reduce the impact of change in depended upon elements on dependant elements.
- Prefer low coupling – assign responsibilities so that coupling remain low.
- Minimizes the dependency hence making system maintainable, efficient and code reusable
- Two elements are coupled, if – One element has aggregation/composition association with another element. – One element implements/extends other element.

SOFTWARE DESIGN

Low Coupling Example



here class RentVideo knows about both VideoStore and Video objects. Rent is depending on both the classes.



VideoStore and Video class are coupled, and Rent is coupled with VideoStore. **Thus providing low coupling**.

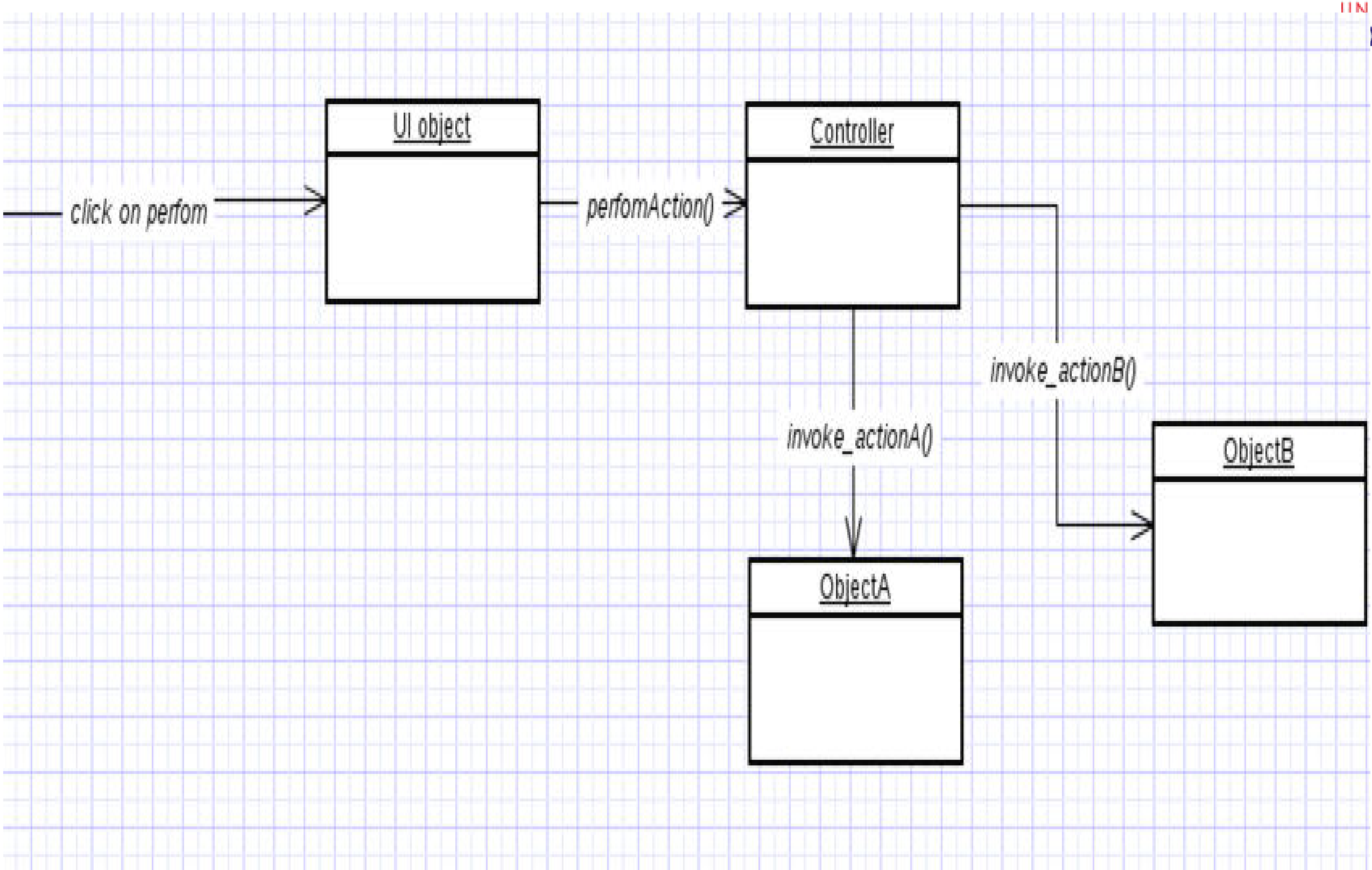
SOFTWARE DESIGN

Controller

- Deals with how to delegate the request from the UI layer objects to domain layer objects .
- Controller helps in minimizing the dependency between GUI components and the system operation classes which represent a set of system level operation
- When a request comes from UI layer object, Controller as a pattern helps us in determining what is that first object that should receive the message from the UI layer objects.
- This object is called controller object which receives request from UI layer object and then controls/coordinates with other object of the domain layer to fulfill the request.
- It delegates the work to other class and coordinates the overall activity.
- Benefits
    - can reuse this controller class.
    - Can use to maintain the state of the use case.
    - Can control the sequence of the activities

SOFTWARE DESIGN

Controller Example



High Cohesion

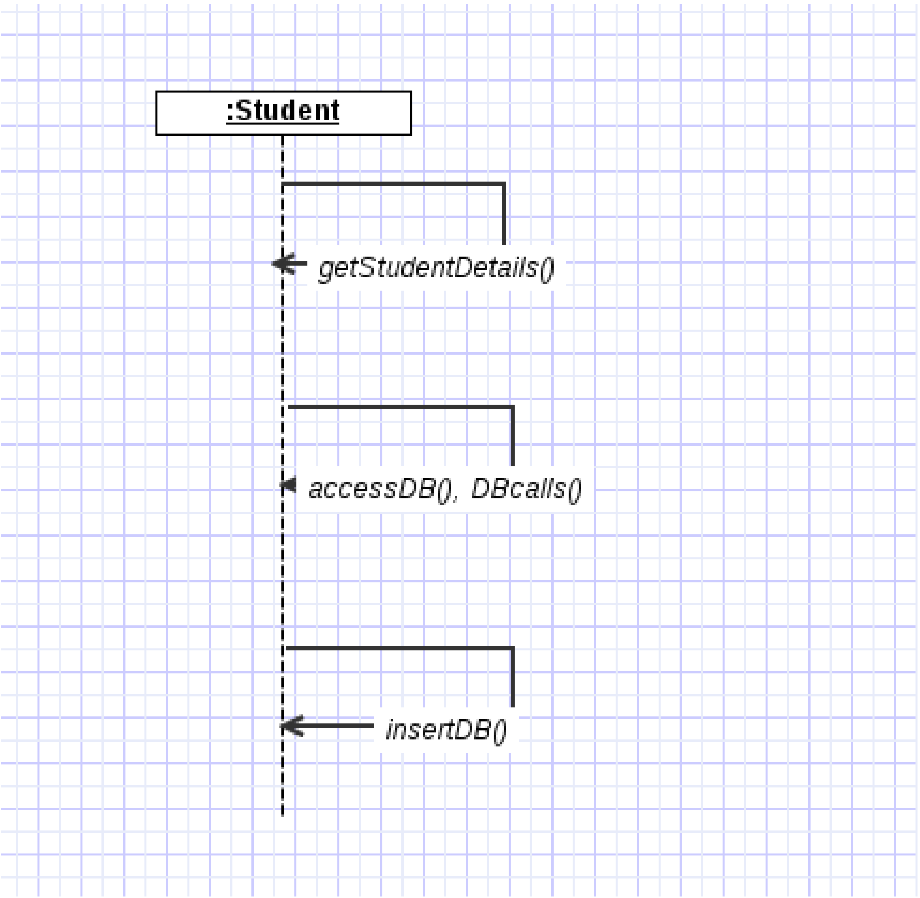
- How are the operations of any element are functionally related?
- Related responsibilities into one manageable unit.
- Prefer high cohesion
- Clearly defines the purpose of the element

Benefits:

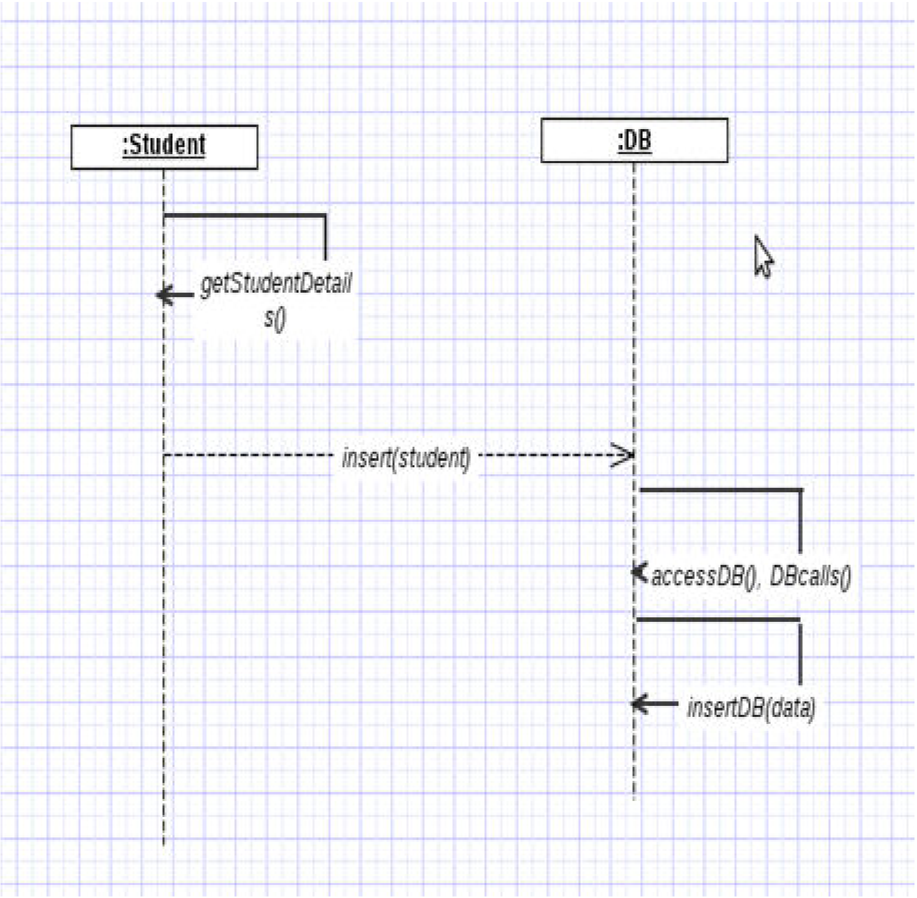
- Easily understandable and maintainable.
- Code reuse
- Low coupling

High Cohesion Example

Low Cohesion



Higher Cohesion



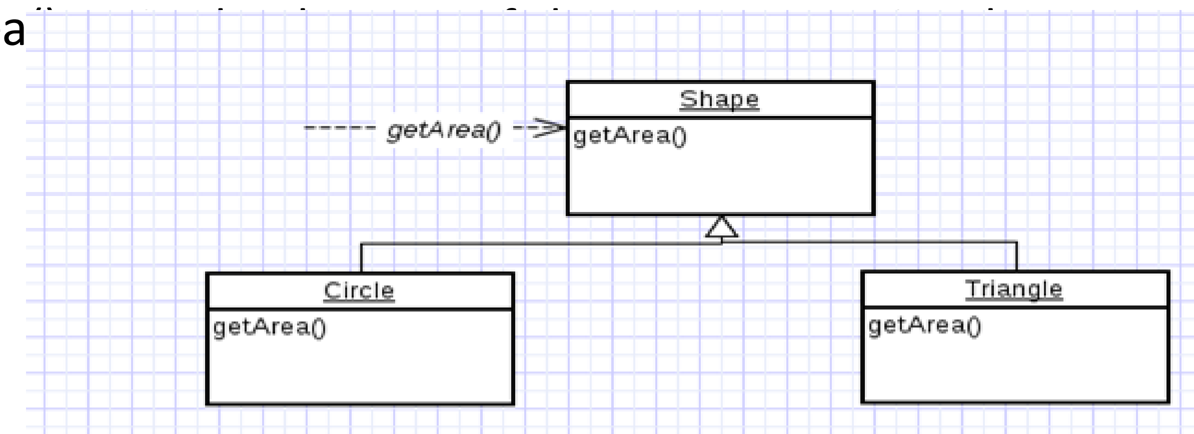
Polymorphism

- How to handle related but varying elements based on element type?
- Polymorphism guides us in deciding which object is responsible for handling those varying elements.

Benefits: Handling new variations will become easy.

Example:

- The getArea … sibility to the subclasses



- By sending message to the Shape object, a call will be made to the corresponding sub class object – Circle or Triangle

Pure Fabrication

- Use of a pure fabrication class that does not represent a concept in the problem domain, to make up a assigned set of related responsibilities.
- Provides a highly cohesive set of activities.
- Behavioral decomposed – implements some algorithm.

  Examples: Adapter, Strategy design patterns
- **Benefits:** High cohesion, low coupling and can reuse this class

Eg.

- Suppose we have a class for Shape and if we must store the shape data in a database.
- If we put this responsibility in Shape class, there will be many database related operations thus making Shape incohesive.
- So, create a fabricated class DBStore which is responsible to perform all database operations.

Eg. logInterface which is responsible for logging information is also a good example for Pure Fabrication.

Indirection

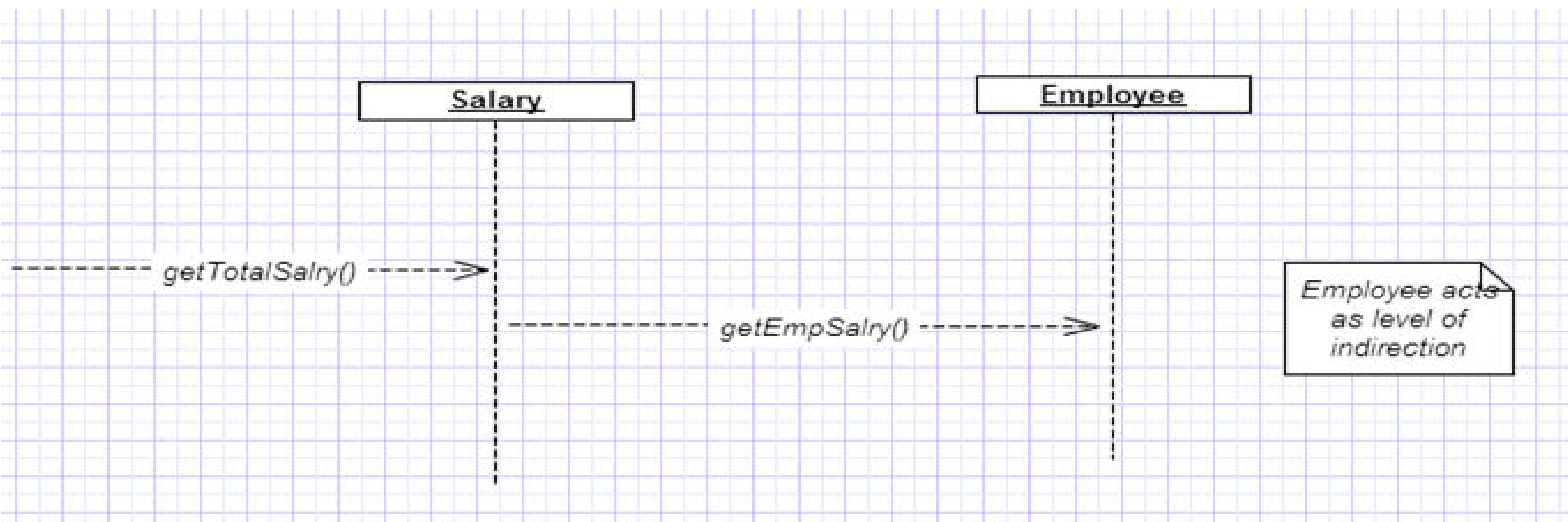
- How can we avoid a direct coupling between two or more elements.
- Indirection introduces an intermediate unit to communicate between the other units, so that the other units are not directly coupled.

Benefits: Low coupling

Design Patterns : Adapter, Façade, Observer

Example : Here polymorphism illustrates indirection

- Class Employee provides a level of indirection to other units of the system.



Protected Variation

- How to avoid impact of variations of some elements on the other elements.
- Achieved by providing a well defined interface so that the there will be no affect on other units and then using polymorphism to create various implementations of this surface
- Provides flexibility and protection from variations.
- Provides more structured design.

Example: Interfaces-polymorphism

Usage

Problem: How to design objects, subsystems, and systems so that the variations or instability in these elements does not have an undesirable impact on other elements?
Solution: Identify points of predicted variation or instability; assign responsibilities to create a stable interface around them.

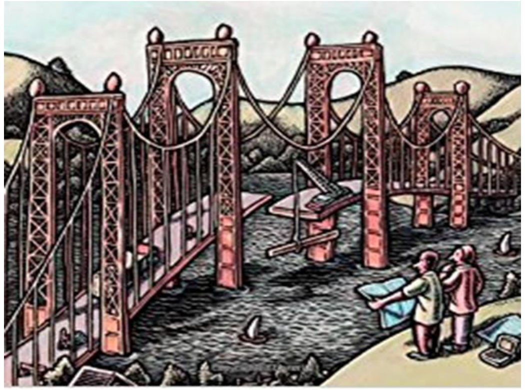
THANK YOU

Dr. H. L. Phalachandra
Department of Computer Science and Engineering
phalachandra@pes.edu

OOAD and SE

Software Design Principles

SOLID Principles



Dr. H.L. Phalachandra
Department of Computer Science and Engineering
leveraging information from slides of
Prof. Vinay Joshi

Acknowledgements: Significant portions of the information in the slide sets presented through the course in the class, are extracted from the prescribed text books, information from the Internet and supplemented by my experience. Since these are only intended for presentation for teaching within PESU, there was no explicit permission solicited. We would like to sincerely thank and acknowledge that the credit/rights remain with the original authors/creators only

OO Design

- If we need to build Object Oriented systems, which are maintainable, extensible and modular, it would be best done following well known principles and practices as part of the OO design.
- These principles will ensure good practices, and when followed will provide loosely coupled classes which will enable minimizing changes in the code, and helps in making code more reusable, maintainable, flexible and stable.
- Following the principles reduces complexity as the application sizes grow.
- We discussed GRASP (**G**eneral **R**esponsibility **A**ssignment **S**oftware **P**rinciples(or Patterns)), which help in providing guidance on assigning responsibilities to collaborating classes & objects, based on their association and their interactions. There were 9 principles of GRASP: Creator, Information Expert, Low Coupling, Controller, High Cohesion, Indirection, Polymorphism, Protected Variations and pure fabrication

SOLID Principles

SOLID is an acronym of the following five basic object oriented design principles listed below initiated by Robert C. Martin

- **S**ingle Responsibility Principle
- **O**pen – Closed Principle
- **L**iskov Substitution Principle
- **I**nterface segregation Principle
- **D**ependency Inversion Principle

1. Single Responsibility Principle

The Single Responsibility Principle (SRP) states: "**A class should have only one responsibility ".** This would mean there would be only one reason to make changes to the code involved post the initial development, thus making the code robust as its likely to have less side effects during maintenance.

- In the following example class

```
Class Simulation{
  Public LoadSimulationFile()        // loading simulation data
  Public Simulate()                  // performing simulation algorithm -
  Public ConvertParams()
  }                                  // this class is handling 2 responsibilities
```

- Since this class has more than one responsibility it has low cohesion. Single responsibility would have made it to be high cohesion
- Splitting the above class can bring about higher cohesion but would increase the coupling.. The goal is to have minimal coupling and not 0 coupling .. So this splitting would be acceptable.

2. Open-Closed Principle

The open closed principle states ***"A software module (it can be a class or method) should be open for extension but be closed for modification"*** or you could state this as ***"you should be able to extend a classes behavior without modifying it"***

- The open-closed principle can be applied through inheritance or through composition
- With Inheritance, you don't have to touch the class you want to extend if you create a subclass of it. The original class is closed for modification but you can add custom code to your subclass to add new behavior.
- This principle needs to be applied strategically in those conditions where you suspect a piece of code will change in the future and its not expected to be applied in every scenario as most of the code may not change.

2. Open-Closed Principle : An Example using Inheritance

- Consider the classes DataStream, NetworkDataStream and the client. In the class Client, ReadData() comes from the NetworkDataStream
- If we need to extend the functionality of the client class to read from another stream the PCIDataStream as shown below
- Thus in this scenario, the client code will function without any error. The client class knows about the base class, and I can pass an object of any of the two subclasses of DataStream. In this way, the client can read data without knowing the underlying subclass. This is achieved without modifying any existing code.

Base Class inherited to NetworkDataStream

```
Class DataStream{
Public byte[] Read()
}

Class NetworkDataStream:DataStream{
Public byte[] Read(){
//Read from the network
 }
}
```

Client

```
Class Client {
Public void ReadData(DataStream ds){
ds.Read();
 }
}
```

New class inheriting again from the base class

```
Class PCIDataStream:DataStream{

Publc byte[] Read(){
//Read data from PCI
 }
}
```

3. Liskov Substitution Principle

The Liskov Substitution Principle states

Derived classes must be substitutable for their base classes.

To elaborate, let’s consider an example, here is an interface whose listing is given below:

```
Public Interface IDevice{
Void Open();
Void Read();
Void Close();
}
```

- This code represents data acquisition device abstraction.
- Data acquisition devices differ based upon their interface types.
- A data acquisition device can use a USB interface, Network Interface (TCP or UDP), PCI express interface, or any other computer interface.
- Clients of iDevice, however, do not need to know what kind of device they are working with.
- This gives programmers an enormous amount of flexibility to adapt to new devices without changing the code which depends upon the iDevice interface.

SOFTWARE DESIGN

3. Liskov Substitution Principle (Cont.)

Let's say if there were only two concrete classes that implemented iDevice interface shown below:

```
public class PCIDevice:IDevice {
   public void Open(){
   // Device specific opening logic
   }
   public void Read(){
   // Reading logic specific to this device
   }
   public void Close(){
   // Device specific closing logic.
   }


}

public class NetWorkDevice:IDevice{

 public void Open(){
 // Device specific opening logic
 }
 public void Read(){
 // Reading logic specific to this device
 }
 public void Close(){
 // Device specific closing logic.
 }


}
```

3. Liskov Substitution Principle (Cont.)

This approach has one problem with USB device. In USB device when you open the connection, data from the previous connection remains in the buffer. Therefore, upon the first read call to the USB device, data from the previous session is returned. This behavior corrupts data for that particular acquisition session.

USB-based device drivers typically provides a refresh function which clears the buffers in the USB-based acquisition device. How can I implement this feature into my code so that the code change remains minimal?

One simple, but unadvised, solution is to update the code by identifying if you are calling the USB object:

```
public class USBDevice:IDevice{
 public void Open(){
    // Device specific opening logic
 }
 public void Read(){
    // Reading logic specific to this device<br>
 }
public void Close(){
   // Device specific closing logic.
 }
 public void Refresh(){
   // specific only to USB interface Device
 }
}

```

```
//Client code..

Public void Acquire(IDevice aDevice){
        aDevice.Open();
        // Identify if the object passed here is USBDevice class Object.
if(aDevice.GetType() == typeof(USBDevice)){
USBDevice aUsbDevice = (USBDevice) aDevice;
aUsbDevice.Refresh();
}

        // remaining code….
}
```

The client code is directly using the concrete class as well as the interface. So abstraction is not sufficient

3. Liskov Substitution Principle (Cont.)

Below is a solution to this problem that follows the LSP:

```
Public Interface IDevice{
Void Open();
Void Refresh();
Void Read();
Void Close();

}
```

Now the client of iDevice is:

```
Public void Acquire(IDevice aDevice)
{
aDevice.open();
aDevice.refresh();
aDevice.acquire()
//Remaining code..

}
```

SOFTWARE DESIGN

3. Liskov Substitution Principle (Cont.)

Lets consider the example of the rectangle and the square. When looking at it from the outset it looks like the square is a specialized version of the rectangle and we may draw the following inheritance hierarchy:

```
Public class Rectangle{
Public void SetWidth(int width){}
Public void SetHeight(int height){}
}

Public Class Square:Rectangle{
//


}
```

- Unfortunately, you cannot substitute a square object in place of a rectangular object. Since the square inherits from the rectangle, therefore, it inherits its method setWidth() and setHeight().
- A client of the square object can change its width and height to different values.
- Since width and height of a square is the same, this may fail the normal behavior of the software.
- So when defining classes, relationships have to be looked at according to different usage scenarios and conditions

4. Interface Segregation Principle (ISP)

The Interface Segregation Principle (ISP) states:

Clients should not be forced to depend upon the interfaces that they do not use.

“Make fine grained interfaces that are client specific”

This principle is looking to indicate

- Make small and specific interfaces so the client who implements them does not depend on methods it does not need.
- Like Instead of having one interface class that can handle three special cases it is better to have three interface, one for each special case.
- This is similar to Classes which should be as specialized as possible, like you do not want any god classes that contain the whole application logic or the source code needing to be modular with every class containing only the minimum necessary logic to achieve the desired behavior.

4. Interface Segregation Principle (ISP) (Contd.)

Again consider the previous example:

```
Public Interface IDevice{
Void Open();
Void Read();
Void Close();
}
```

```
 Public Interface IDevice{
Void Open();
        Void Refresh();
Void Read();
Void Close();
}
```

For example, the following lines of code need to be added to the Network Device class and PCI Device class to work with this design:

```
public  void Refresh()
{
// Yes nothing here… just a useless blank function
}
```

This breaks this principle as it is causing unnecessary clients to depend on the same.

4. Interface Segregation Principle (ISP)

One approach to solve is to use the understanding that refresh occurs after the open function only in USB interface. Thus

- refresh logic is moved from the client of IDevice to the specific USB concrete device class as shown here:

```
Public Interface IDevice{
Void Open();
Void Read();
Void Close();
}


Public class USBDevice:IDevice{
Public void Open{
// open the device here…

// refresh the device
this.Refresh();

 }
Private void Refresh(){
// make the USb Device Refresh
 }

}
```

- Thus this reduces the number of functions in the iDevices class and makes it less fat too.

5. Dependency Inversion Principle (DIP)

- This principle is a generalization of the other principles discussed above.

Program to an interface, not to an implementation.

Consider following example

```
Class PCIDevice{
Void open(){}
Void close(){}
}
Static void Main(){
PCIDevice aDevice = new PCIDevice();
aDevice.open();
//do some work
aDevice.close();

}
```

This violates the “program to an interface" principle because we are working with the reference of the concrete class PCI Device.

The below listing addresses this

```
Interface IDevice{
Void open();
Void close();
}
Class PCIDevice implements IDevice{
Void open(){ // PCI device opening code }
Void close(){ // PCI Device closing code }
}
Static void Main(){
IDevice aDevice = new PCIDevice();
aDevice.open();
//do some work
aDevice.close();
}
```

5. Dependency Inversion Principle (DIP) (Cont.)

High-level modules should not depend upon low-level modules. Both should depend upon abstractions.

Consider the following code

```
Class TransferManager{
public void TransferData(USBExternalDevice usbExternalDeviceObj,SSDDrive  ssdDriveObj){
            Byte[] dataBytes = usbExternalDeviceObj.readData();

           // work on dataBytes e.g compress, encrypt etc..

            ssdDriveObj.WrtieData(dataBytes);
        }
}

Class USBExternalDevice{
Public byte[] readData(){
        }
}

Class SSDDrive{
Public void WriteData(byte[] data){
}
}
```

Say, this is for transferring data from a source (external device) to a storage device (internal device). In this code, there are three classes.

- The Transfer Manager class represents a ***high-level module*** as it uses two classes in its one function. Hence the other two classes are ***low-level modules***.
- The high-level module function (Transfer Data) defines logic based upon how data is transferred from one device to another device.
- Any module which is controlling the logic, and uses the low-level modules in doing so, is called the **high-level module**.
- In the code above, the high-level module is directly (without any abstraction) using the lower-level modules, hence violating the dependency inversion principle.
- if you want to add other external devices you will have to change the higher-level module. Hence your higher-level module will be dependent upon the lower-level module, and that dependency will make the code difficult to change.

5. Dependency Inversion Principle (DIP) (Cont.)

The solution is easy if use the principle above: "program to an interface." Here is the listing:

```
Class USBExternalDevice implements IExternalDevice{
Public byte[] readData(){
}
}

Class SSDDrive implements IInternalDevice{
Public void WriteData(byte[] data){
}
}

Class TransferManager implements ITransferManager{
public void Transfer(IExternalDevice externalDeviceObj, IInternalDevice internalDeviceObj){
           Byte[] dataBytes = externalDeviceObj.readData();

           // work on dataBytes e.g compress, encrypt etc..

           internalDeviceObj.WrtieData(dataBytes);
        }
}

Interface IExternalDevice{
        Public byte[] readData();
}

Interfce IInternalDevice{
Public void WriteData(byte[] data);
}

Interface ITransferManager {
public void Transfer(IExternalDevice usbExternalDeviceObj,SSDDrive  IInternalDevice);
}
```

Since in the code, both high level module and the low level module depend upon abstraction, this is following the dependency inversion principle

THANK YOU

Dr. H. L. Phalachandra
Department of Computer Science and Engineering
phalachandra@pes.edu

SOFTWARE ENGINEERING

SOFTWARE IMPLEMENTATION

Prof. Phalachandra H. L
Department of Computer Science and Engineering

Acknowledgements: Significant portions of the information in the slide sets presented through the course in the class, are extracted from the prescribed text books, information from the Internet and supplemented by my experience. Since these are only intended for presentation for teaching within PESU, there was no explicit permission solicited. We would like to sincerely thank and acknowledge that the credit/rights remain with the original authors/creators only

SOFTWARE IMPLEMENTATION

Introduction, Implementation and Code Characteristics

Prof. Phalachandra H. L
Department of Computer Science and Engineering

SOFTWARE IMPLEMENTATION

Context

- Implementation phase in an SDLC starts post the Requirements, Architecture and Design, and converts the program structures developed as part of Design to actual code using some programming language.
- The level of details needed before implementation can start, is dependent on whether you are using a compiled language or an interpreted language.
- This will involve choice of a language, choice of a development environment, following of a configuration management plan and building code.
- As part of building code, this would need the usage of Coding standards, Coding guidelines and following the language syntaxes in for translating designs to code.
- Given that code can exist for a long time there are number of practices followed as part of coding to address potential concerns regarding quality, security and testability.
- The entry criteria for this phase is baselined Design (Detailed design) and the exit criteria for this phase is unit tested, peer reviewed functioning code.

Software Construction Introduction

Definition

Software Construction or implementation is the detailed creation of working software through a combination of coding, reviews and unit testing.

Some of the fundamental goals when constructing or implementing are

- Minimizing complexity
- Anticipating change
- Constructing for verification
- Reuse
- Construction for others to read

- Implementation has a development environment, whether integrated or discrete, a structure which is designed for holding your code and an Software Configuration Management system driving the management of the same

Characterizing Software Construction

- **Construction produces high volume of configuration items**
  - E.g., source files, test cases etc.
- **Construction is tool-intensive**
  - relies heavily on tools such as compilers, debuggers, GUI builders etc.
- **Construction is related to software quality**
  - code is the ultimate deliverable of a software project
- **Construction makes extensive use of computer science knowledge**
  - in using algorithms, detailed coding practices etc.

SOFTWARE IMPLEMENTATION

Choice of a Programming Language based on levels of Abstraction

- **Assembly languages** map directly onto the CPU architecture.
- **Procedural languages** provide a modest level of abstraction from the underlying computer.
- **Aspect-orientation supported by languages** are designed to allow the developer to separate aspects (or concerns) within a program.
- **Object-oriented languages** break away from CPU architecture and further abstract the machine by enabling the designer and the developer to consider the problem in terms of objects.

Development Environment - 1

- **Choice in selecting development & implementation environments**
  - restricted by the choice of language, methodology, application, and development and target hardware platforms
- **Some factors in selecting development environments:**
  - **Commercial vs. open-source**
    - Developers often prefer open-source tools, but some organizations think that open-source tools are untrustworthy or less capable than commercial tools
  - **Support of development process**
    - Support of the development process includes ensuring that tools such as debuggers, test builders, and analysis tools are available

Coding

- Coding activity begins once the Development environment is chosen.

Prominent Quotes of Kernighan

"Where there are two bugs, there is likely to be a third.“

“Don't patch bad code - rewrite it."

"Don't stop with your first draft."

Coding : Journey of a Bug

Coding activity begins once the Development environment is chosen.

Journey of a bug Starts here

Consider the following piece of (buggy) code:

Most Likely

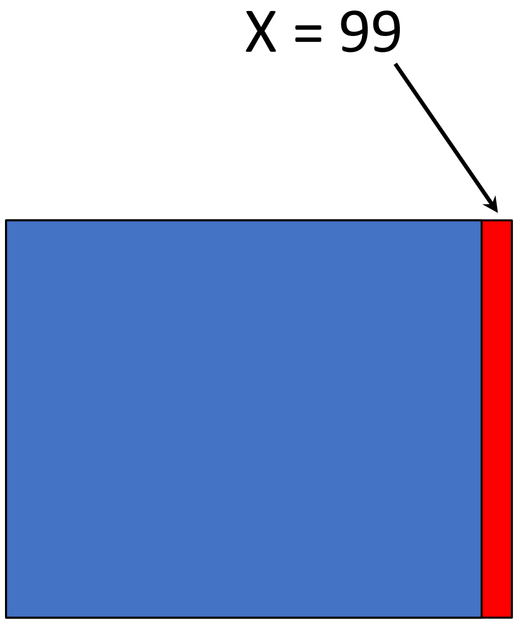
```
x <= 99
```

```
if (x<99)
   process_event();
else ???
```

Not specified System can misbehave

Bug Travels !!

- Our above code could be used in a touch panel driver
  - Industrial, automotive, aeronautics …
- What does our bug do?
  - It makes the right edge to go undetected

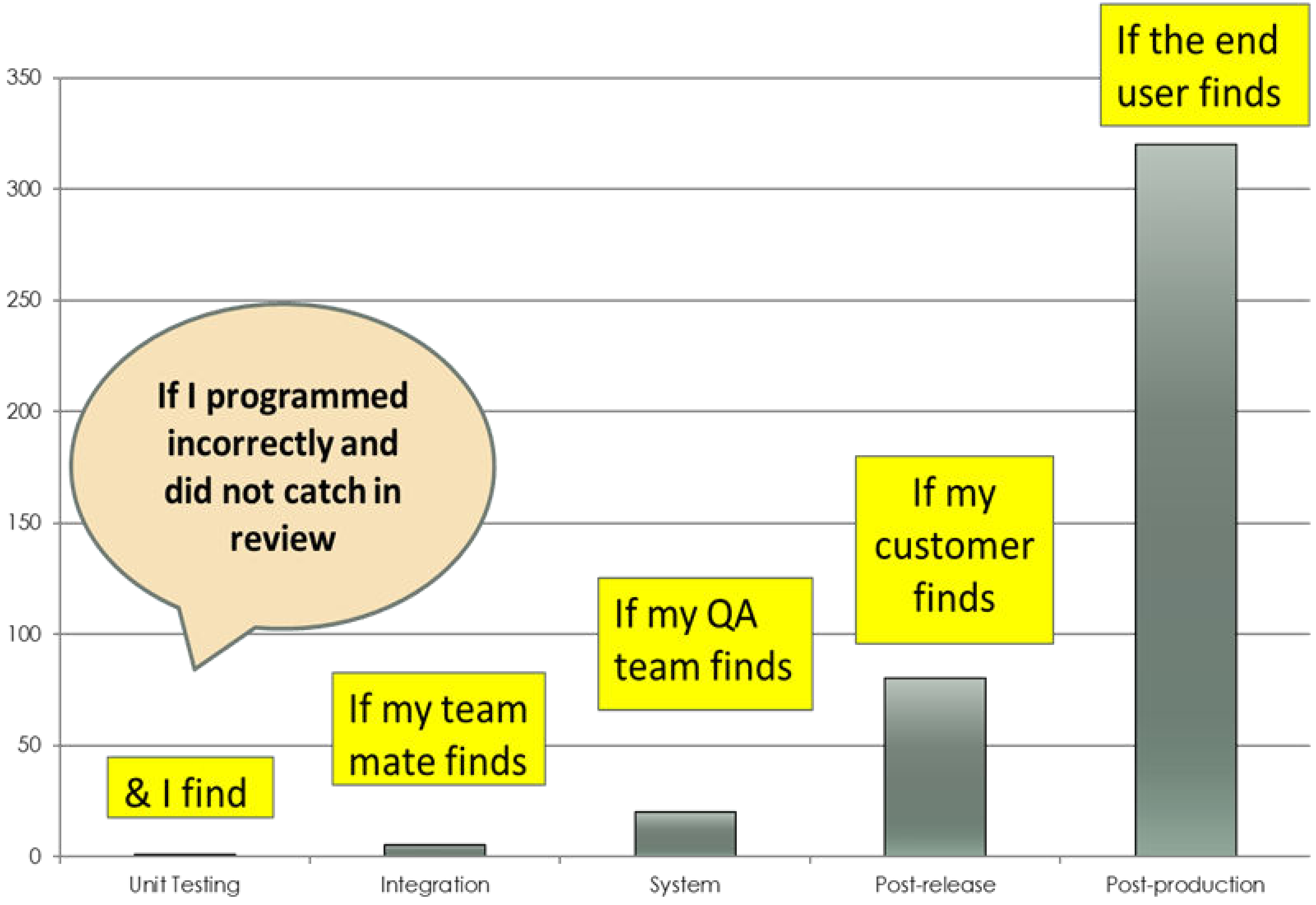


Where can we catch the Bug

- Coding – before the bug occurs!
- Code review
- Unit testing
- Integration testing
- System testing
- Field trials
- Production

SOFTWARE IMPLEMENTATION

How expensive is the Bug



Keep these thoughts in mind while programming

- A program is for the programmer who reads it, not for the computer that runs it.
- A program is written once, but read and modified a lot many times.
- Assume that the person who will end up maintaining your code is a psychopath who knows where you live 
- Minimize the complexity of code which you write
- When reviewing code, count to 5 before concluding, 10 before refactoring

Characteristics of Code

Some of the different characteristics of code are

- Programs should be Simple and Clear
- Naming
- Structuring
  - File Structure
  - Code and Data Structure
- Ease of reading and Understanding
  - Whitespaces
  - New Lines
  - Indents
  - Commenting

Well named, structured, easy to understand code is easy to verify, debug, test, maintain and reuse

Code Characteristics : Programs should be Simple and Clear

Programs should be well thought out, well designed, well structured although

Programmer's have a tendency to jump start

"Not too many"

- Lines per function
- Lines or functions per file
- Arguments per function
- Levels of nesting
- Conditions

Code Characteristics : Naming & Naming Conventions

Naming is important! and have to be **Meaningful** and should not be close to keywords

What is in a name? A rose by any other name would smell as sweet.

- **William Shakespeare** **… BUT**

Its good to have **descriptive names**. Although it might be ok to use short names for local variables, its good to use 2-3 character short names

Naming could be done as

- **Pascal Casing** (first letter upper case other lower case) e.g. BackColor for class names/Variable Names/ method parameters
- **Camel Casing** (first letter upper case others lower case except 1st word) e.g. backColor for method names
- **Underscore Prefix** (_underScore) For underscore (__), the word after _ use camelCase terminology

Code Characteristics : Naming convention – Examples

What does the following code do?

```
int uglyduck (int timer_count)
{
   if (timer_count <= 1)
      return 1;
   else
      return timer_count * uglyduck(timer_count-1);
}
```

Need meaningful Names

```
for (i = 0; i < nelems; i++)
   sum += elems[i];

for (iElementIndex = 0;
    iElementIndex < iNumOfElements;
    iElementIndex++)
{
   sumOfElements += ElementArray[iElementIndex];
}
```

With Descriptive Names

Code Characteristics : Other Naming conventions

There are other naming conventions to maximize code portability across compilers like

- No variable shall have a name that is a keywords of programming languages
- No variable shall have a name that starts with an underscore
- No variable name shall be longer than 31 characters
- No variable name shall contain any numeric value that is called out elsewhere
- Each variable name shall be descriptive of its purpose
- …….

Code Characteristics : Structuring

Structure refers to the dependencies between software components (subprograms, parameters, code blocks, etc). Dependencies can be highlighted through proper naming and layout. For example, dependent subprogram interface/implementation declarations should be closer to each other.

File structure

- Logically grouped

  #includes, #defines, structures, functions

- Well partitioned

  What goes into a header file? What goes into a C file?

  What functions/variables should be exposed?

Code and data structure

- Everything that you learnt in ADT!
- Well encapsulated, logically grouped.
- Properly initialized!

Code Characteristics : Ease of Reading and Understanding

Readability is dependent upon identifier naming and the visual layout of statements. For example, use standardized indentation and blank lines/spaces to illustrate blocks of code and their hierarchical dependencies.

Addition of Spaces

E.g. `for(int i=0;i<40;i++)`

```
for( int i = 0; i < 40; i++)
```

How is this compared to The earlier one

Code Characteristics : Ease of Reading and Understanding

Addition of new lines

```
for (i = 0; i < len; i++) if (num == array[i]) break;
if (i == len) return -1; else return i;
```

```
for (i = 0; i < len; i++)
        if (num == array[i])
            break;
if (i == len)
        return -1;
else
        return i;
```

Use of Parenthesis

E.g. if (c >= ‘0’ && c <= ‘9’ || c >= ‘a’ && c <= ‘f’ || c >= ‘A’ && c <= ‘F’)

Soln -

SOFTWARE IMPLEMENTATION

Code Characteristics : Ease of Reading and Understanding

Comments for code should explain and clarify the program, rather then simply repeat code. Comments should concisely and clearly explain the underlying logic. They should not be cryptic or misleading, and must be easy to identify using proper indentation, spacing and 'highlighting' (e.g., comment boxes).

Well Commented

```
/* Enable interrupts, start Rx and Tx. */
CFGREG = 0x1003;
```

vis-a-vis

```
/* Check if the pointer is NULL. */
if (alloc_ptr == NULL)
    panic ();
```

Which is very obvious and redundant

Do’s of good programming style

1. Use a few standards, agreed upon control constructs.
2. Use GOTO in a disciplined way.
3. Use user-defined data types to model entities in the problem domain.
4. Hide data structure behind access functions
5. Use appropriate variable names
6. Use indentation, parentheses, blank spaces, and blank lines to enhance readability.

Don’ts of good programming style

1. Don’t be too clever.
2. Avoid null Then statement
3. Avoid Then If statement
4. Don’t nest too deeply.
5. Don’t use an identifier for multiple purposes.
6. Examine routines having more than five formal parameters.

THANK YOU

Prof. Phalachandra H.L.
Department of Computer Science and Engineering
phalachandra@pes.edu

Refactoring Code

- Refactoring consists of improving the internal structure of an existing program’s source code, while preserving its external behavior.
- Refactoring is intended to improve the
  - Design, structure, and/or implementation of the software (its non-functional attributes), while preserving its functionality.
  - Refactoring improves objective attributes of code (length, duplication, coupling and cohesion, cyclomatic complexity) that correlate with ease of maintenance
  - refactoring helps code understanding
- Refactoring does “not” mean:
  - rewriting code
  - fixing bugs
  - improving observable aspects of software such as its interface

SOFTWARE ENGINEERING

SOFTWARE IMPLEMENTATION

Prof. Phalachandra H. L
Department of Computer Science and Engineering

Acknowledgements: Significant portions of the information in the slide sets presented through the course in the class, are extracted from the prescribed text books, information from the Internet and supplemented by my experience. Since these are only intended for presentation for teaching within PESU, there was no explicit permission solicited. We would like to sincerely thank and acknowledge that the credit/rights remain with the original authors/creators only

SOFTWARE IMPLEMENTATION

Introduction, Implementation and Code Characteristics

Prof. Phalachandra H. L
Department of Computer Science and Engineering

Context

- Implementation phase in an SDLC starts post the Requirements, Architecture and Design, and converts the program structures developed as part of Design to actual code using some programming language.
- The level of details needed before implementation can start, is dependent on whether you are using a compiled language or an interpreted language.
- This will involve choice of a language, choice of a development environment, following of a configuration management plan and building code.
- As part of building code, this would need the usage of Coding standards, Coding guidelines and following the language syntaxes in for translating designs to code.
- Given that code can exist for a long time there are number of practices followed as part of coding to address potential concerns regarding quality, security and testability.
- The entry criteria for this phase is baselined Design (Detailed design) and the exit criteria for this phase is unit tested, peer reviewed functioning code.

Definition

Software Construction or implementation is the detailed creation of working software through a combination of coding, reviews and unit testing.

Some of the fundamental goals when constructing or implementing are

- Minimizing complexity
- Anticipating change
- Constructing for verification
- Reuse
- Construction for others to read

- Implementation has a development environment, whether integrated or discrete, a structure which is designed for holding your code and an Software Configuration Management system driving the management of the same

Characterizing Software Construction

- **Construction produces high volume of configuration items**
  - E.g., source files, test cases etc.
- **Construction is tool-intensive**
  - relies heavily on tools such as compilers, debuggers, GUI builders etc.
- **Construction is related to software quality**
  - code is the ultimate deliverable of a software project
- **Construction makes extensive use of computer science knowledge**
  - in using algorithms, detailed coding practices etc.

Choice of a Programming Language based on levels of Abstraction

- **Assembly languages** map directly onto the CPU architecture.
- **Procedural languages** provide a modest level of abstraction from the underlying computer.
- **Aspect-orientation supported by languages** are designed to allow the developer to separate aspects (or concerns) within a program.
- **Object-oriented languages** break away from CPU architecture and further abstract the machine by enabling the designer and the developer to consider the problem in terms of objects.

Development Environment - 1

- **Choice in selecting development & implementation environments**
  - restricted by the choice of language, methodology, application, and development and target hardware platforms
- **Some factors in selecting development environments:**
  - **Commercial vs. open-source**
    - Developers often prefer open-source tools, but some organizations think that open-source tools are untrustworthy or less capable than commercial tools
  - **Support of development process**
    - Support of the development process includes ensuring that tools such as debuggers, test builders, and analysis tools are available

Coding

- Coding activity begins once the Development environment is chosen.

Prominent Quotes of Kernighan

"Where there are two bugs, there is likely to be a third.“

“Don't patch bad code - rewrite it."

"Don't stop with your first draft."

Coding : Journey of a Bug

Coding activity begins once the Development environment is chosen.

Journey of a bug Starts here

Consider the following piece of (buggy) code:

Most Likely

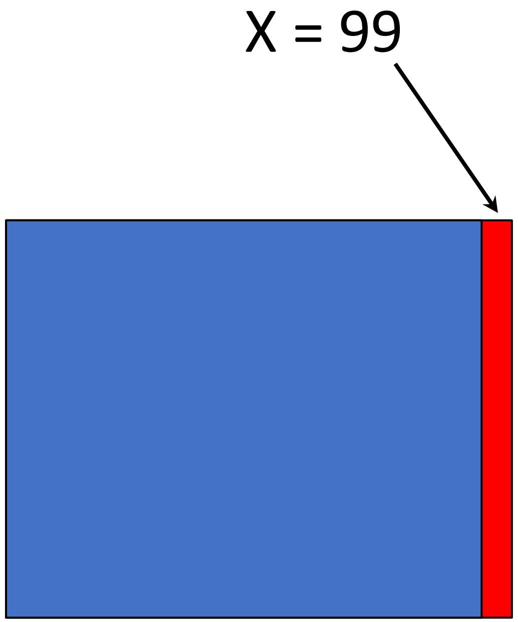
```
x <= 99
```

```
if (x<99)
   process_event();
else ???
```

Not specified System can misbehave

Bug Travels !!

- Our above code could be used in a touch panel driver
  - Industrial, automotive, aeronautics …
- What does our bug do?
  - It makes the right edge to go undetected

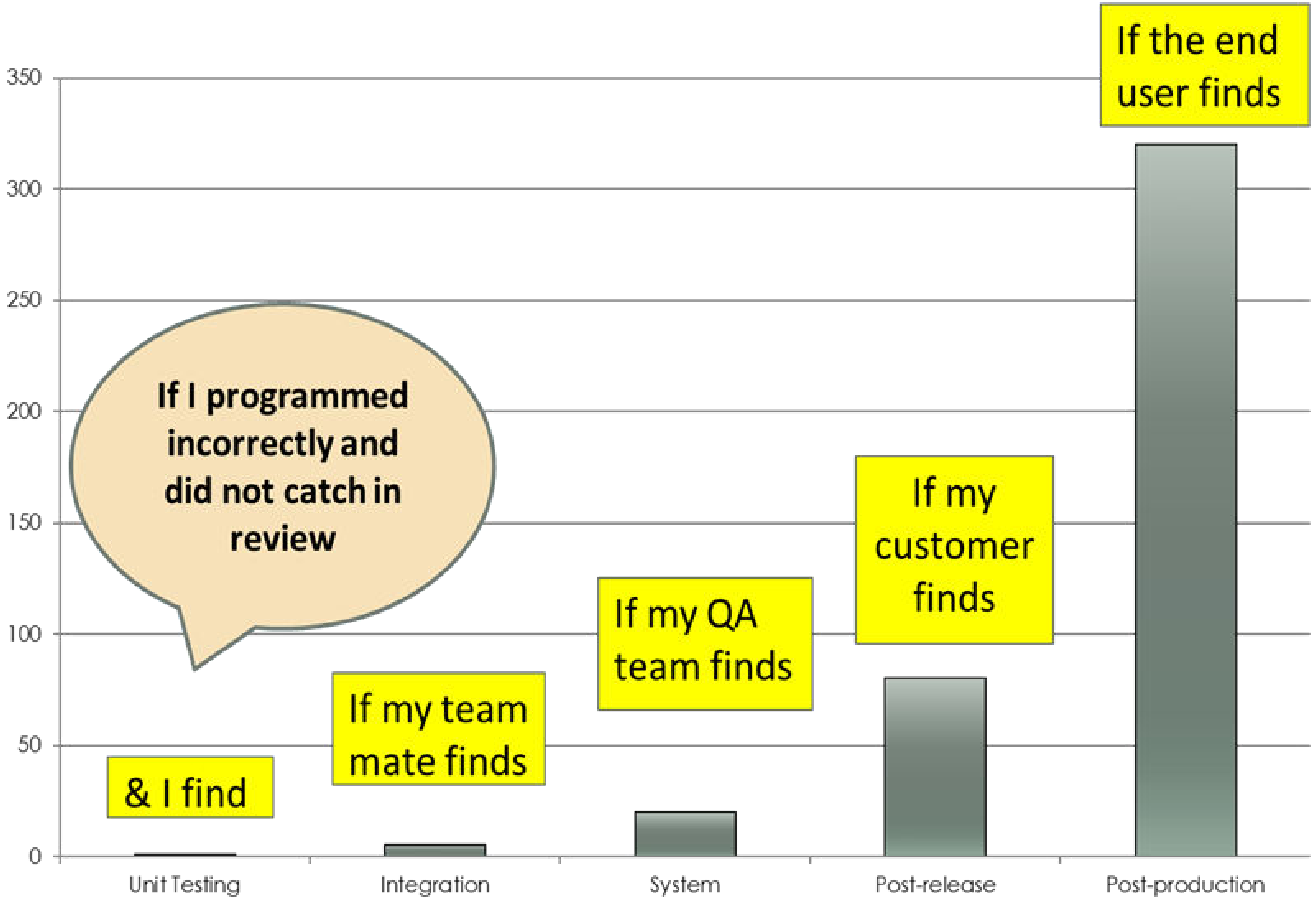


Where can we catch the Bug

- Coding – before the bug occurs!
- Code review
- Unit testing
- Integration testing
- System testing
- Field trials
- Production

SOFTWARE IMPLEMENTATION

How expensive is the Bug



SOFTWARE IMPLEMENTATION

Keep these thoughts in mind while programming

- A program is for the programmer who reads it, not for the computer that runs it.
- A program is written once, but read and modified a lot many times.
- Assume that the person who will end up maintaining your code is a psychopath who knows where you live

- Minimize the complexity of code which you write
- When reviewing code, count to 5 before concluding, 10 before refactoring

Characteristics of Code

Some of the different characteristics of code are

- Programs should be Simple and Clear
- Naming
- Structuring
  - File Structure
  - Code and Data Structure
- Ease of reading and Understanding
  - Whitespaces
  - New Lines
  - Indents
  - Commenting

Well named, structured, easy to understand code is easy to verify, debug, test, maintain and reuse

Code Characteristics : Programs should be Simple and Clear

Programs should be well thought out, well designed, well structured although

Programmer's have a tendency to jump start

"Not too many"

- Lines per function
- Lines or functions per file
- Arguments per function
- Levels of nesting
- Conditions

Code Characteristics : Naming & Naming Conventions

Naming is important! and have to be **Meaningful** and should not be close to keywords

What is in a name? A rose by any other name would smell as sweet.

- **William Shakespeare** **… BUT**

Its good to have **descriptive names**. Although it might be ok to use short names for local variables, its good to use 2-3 character short names

Naming could be done as

- **Pascal Casing** (first letter upper case other lower case) e.g. BackColor for class names/Variable Names/ method parameters
- **Camel Casing** (first letter upper case others lower case except 1st word) e.g. backColor for method names
- **Underscore Prefix** (_underScore) For underscore (__), the word after _ use camelCase terminology

Code Characteristics : Naming convention – Examples

What does the following code do?

```
int uglyduck (int timer_count)
{
   if (timer_count <= 1)
      return 1;
   else
      return timer_count * uglyduck(timer_count-1);
}
```

Need meaningful Names

```
for (i = 0; i < nelems; i++)
   sum += elems[i];

for (iElementIndex = 0;
    iElementIndex < iNumOfElements;
    iElementIndex++)
{
   sumOfElements += ElementArray[iElementIndex];
}
```

With Descriptive Names

Code Characteristics : Other Naming conventions

There are other naming conventions to maximize code portability across compilers like

- No variable shall have a name that is a keywords of programming languages
- No variable shall have a name that starts with an underscore
- No variable name shall be longer than 31 characters
- No variable name shall contain any numeric value that is called out elsewhere
- Each variable name shall be descriptive of its purpose
- …….

Code Characteristics : Structuring

Structure refers to the dependencies between software components (subprograms, parameters, code blocks, etc). Dependencies can be highlighted through proper naming and layout. For example, dependent subprogram interface/implementation declarations should be closer to each other.

File structure

- Logically grouped

  #includes, #defines, structures, functions

- Well partitioned

  What goes into a header file? What goes into a C file?

  What functions/variables should be exposed?

Code and data structure

- Everything that you learnt in ADT!
- Well encapsulated, logically grouped.
- Properly initialized!

Code Characteristics : Ease of Reading and Understanding

Readability is dependent upon identifier naming and the visual layout of statements. For example, use standardized indentation and blank lines/spaces to illustrate blocks of code and their hierarchical dependencies.

Addition of Spaces

E.g. for(int i=0;i<40;i++)

```
for( int i = 0; i < 40; i++)
```

How is this compared to The earlier one

Code Characteristics : Ease of Reading and Understanding

Addition of new lines

```
for (i = 0; i < len; i++) if (num == array[i]) break;
if (i == len) return -1; else return i;
```

```
for (i = 0; i < len; i++)
        if (num == array[i])
            break;
if (i == len)
        return -1;
else
        return i;
```

Use of Parenthesis

E.g. if (c >= ‘0’ && c <= ‘9’ || c >= ‘a’ && c <= ‘f’ || c >= ‘A’ && c <= ‘F’)

Soln -

Code Characteristics : Ease of Reading and Understanding

Comments for code should explain and clarify the program, rather then simply repeat code. Comments should concisely and clearly explain the underlying logic. They should not be cryptic or misleading, and must be easy to identify using proper indentation, spacing and 'highlighting' (e.g., comment boxes).

Well Commented

```
/* Enable interrupts, start Rx and Tx. */
CFGREG = 0x1003;
```

vis-a-vis

```
/* Check if the pointer is NULL. */
if (alloc_ptr == NULL)
    panic ();
```

Which is very obvious and redundant

Do’s of good programming style

1. Use a few standards, agreed upon control constructs.
2. Use GOTO in a disciplined way.
3. Use user-defined data types to model entities in the problem domain.
4. Hide data structure behind access functions
5. Use appropriate variable names
6. Use indentation, parentheses, blank spaces, and blank lines to enhance readability.

Don’ts of good programming style

1. Don’t be too clever.
2. Avoid null Then statement
3. Avoid Then If statement
4. Don’t nest too deeply.
5. Don’t use an identifier for multiple purposes.
6. Examine routines having more than five formal parameters.

THANK YOU

Prof. Phalachandra H.L.
Department of Computer Science and Engineering
phalachandra@pes.edu

Refactoring Code

- Refactoring consists of improving the internal structure of an existing program's source code, while preserving its external behavior.
- Refactoring is intended to improve the
  - Design, structure, and/or implementation of the software (its non-functional attributes), while preserving its functionality.
  - Refactoring improves objective attributes of code (length, duplication, coupling and cohesion, cyclomatic complexity) that correlate with ease of maintenance
  - refactoring helps code understanding
- Refactoring does "not" mean:
  - rewriting code
  - fixing bugs
  - improving observable aspects of software such as its interface

SOFTWARE ENGINEERING

SOFTWARE IMPLEMENTATION

Prof. Phalachandra H. L
Department of Computer Science and Engineering

Acknowledgements: Significant portions of the information in the slide sets presented through the course in the class, are extracted from the prescribed text books, information from the Internet and supplemented by my experience. Since these are only intended for presentation for teaching within PESU, there was no explicit permission solicited. We would like to sincerely thank and acknowledge that the credit/rights remain with the original authors/creators only

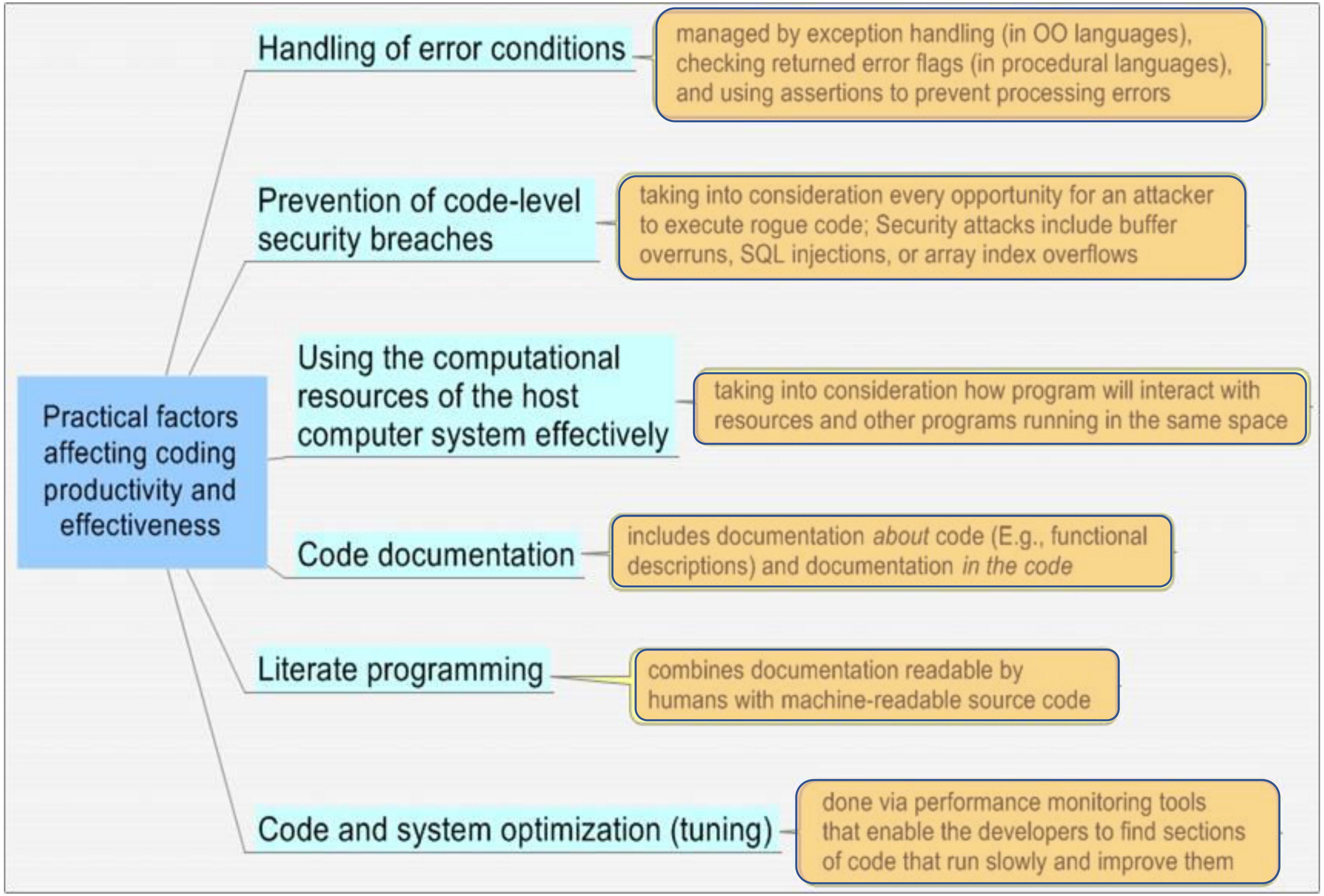
SOFTWARE IMPLEMENTATION

Factors for coding

(Practical Considerations, Standards & Guidelines)

Prof. Phalachandra H. L
Department of Computer Science and Engineering

Practical Considerations for effective Coding



Coding Standards & Guidelines Generics

- Standards are rules which are mandatory to be followed while guidelines are recommended to be followed.
- Significant literature use these two terminologies interchangeably
- There are rules which can be standards and guidelines.
- These can be practiced as checklists and enforced as part of reviews
- These could be specific for a language, company or a group
- Some examples of Standards and Guidelines
  - Standard headers for different modules
  - Naming Conventions for local and global variables and for constant identifiers should be as ....
  - One declaration per line
  - Avoid copy and paste
  - Avoid magic numbers

Coding Standards – Mandatory to be followed

- A coding standard provides a uniform appearance to the codes written by different engineers
- Using coding standards, improves readability, maintainability and reduces complexity too
- Some of the coding practices are part of coding standards like
  - Defensive programming
  - Secure programming
  - Testable programming
- Thus some of the commonly occurring issues with code can be proactively addressed e.g. by having a standard like All “function/method return values” have to be handled will enable gracious handling.
- It helps in code reuse
- It promotes sound programming practices and increases efficiency of the programmers.

Coding Standards – Mandatory to be followed - Illustrations

1. Rules for usage of global about which type of data
2. Standard headers for different modules containing
   - Name of the module
   - Date of module creation
   - Author of the module
   - Modification history
   - Synopsis of the module about what the module does
   - Different functions supported in the module along with their input output parameters
   - Global variables accessed or modified by the module
3. Naming convention for variables – one type of naming for local variables and one for global data
4. Error return values and exception handling conventions like return 0 or -1
5. …………………

Coding Standard Practices : Defensive Programming

Murphy's Law **-** If anything can go wrong, it will.

- Redundant code is incorporated to check system state after modifications
- Implicit assumptions are tested explicitly

```
pid = fork ();
  if (pid == 0)
    {
      /* child process ... do your stuff */
    }
  else if (pid < 0)
    /* The fork failed.  Report failure.  */
else
    /* This is the parent process.. do your stuff */
 ..
```

Coding Standard Practices : Secure Programming

Secure coding is the practice of developing computer software in a way that guards against the accidental introduction of security vulnerabilities

Defects, bugs and logic flaws are consistently the primary cause of commonly exploited software vulnerabilities. Some of the approaches or practices which can avoid these are

1. **Validate input**. Validate input from all untrusted data sources.
2. **Heed compiler warnings**. Compile code using the highest warning level available for your compiler and eliminate warnings by modifying the code. Use static and dynamic analysis tools to detect and eliminate additional security flaws.
3. **Default deny**. Access decisions are based on permission rather than exclusion or by default, access is denied & identified ones are permitted.

Coding Standard Practices : Secure Programming

4. **Adhere to the principle of least privilege**. Every process should execute with the least set of privileges necessary to complete the job. Elevated permission should only be accessed for the least amount of time required to complete the privileged task.
5. **Sanitize data sent to other systems**. Sanitize all data passed to complex subsystems such as command shells, relational databases, and commercial off-the-shelf (COTS) components.

SOFTWARE IMPLEMENTATION

Coding Standard Practices : Programming for Testability

We need to construct software to make it easy to test … and find and fix bugs

- **Assertions** — Use assertions to identify out-of-range or inappropriate values whenever possible
- **Test points** — methods that can be used to set or retrieve current module status and variable contents
- **Scaffolding** — code written to emulate a feature of the system that the object would call or send a message to
- **Test harness** — code written to drive objects, modules, or components as if they were part of the complete system
- **Test stubs** — code that returns known fixed values that emulate functions not currently under test
- **Instrumenting** — execution logging and "I got here" messages to know what's going on inside modules
- **Building test data sets** — … includes testing both valid and invalid data

Coding Guidelines – Recommended to be followed

- Like coding standards, coding guidelines also gives a uniform appearance to the codes written by different engineers
- Most of these are guidelines are generic suggestions regarding the coding style that need to be followed for betterment of understanding, readability of the code.
- Helps in detecting errors in the early phases, thus reducing the extra cost incurred by the software project.
- Increases readability and understandability thus it reduces the complexity of the code.
- It makes it simpler for others & helps towards maintenance over the lifetime of the product. Eg.

1. Code should be well documented
2. Indent code with spaces and lines
3. Keep the length of the functions small
4. Minimize using GOTO statements
5. Avoid using identifiers for multiple purposes
6. …………………

THANK YOU

Prof. Phalachandra H.L.
Department of Computer Science and Engineering
phalachandra@pes.edu

SOFTWARE ENGINEERING

SOFTWARE IMPLEMENTATION

Prof. Phalachandra H. L

Department of Computer Science and Engineering

Acknowledgements: Significant portions of the information in the slide sets presented through the course in the class, are extracted from the prescribed text books, information from the Internet and supplemented by my experience. Since these are only intended for presentation for teaching within PESU, there was no explicit permission solicited. We would like to sincerely thank and acknowledge that the credit/rights remain with the original authors/creators only

SOFTWARE IMPLEMENTATION

Managing Construction & Tools

Prof. Phalachandra H. L
Department of Computer Science and Engineering

Managing Construction

During the construction of a software, a number of key issues are critical to the overall integrity and functionality of the software solution

- Minimizing complexity
- Anticipating change
- Constructing software solutions for verification
- Constructing software using standards

We have looked at these during the construction

Management of Implementation would involve looking at it from two perspectives

Proceeding as planned?	Technical quality?
Typical measures: effort expenditure and rate of completion	Typical measures: Lines of code developed, number of defects found in testing etc.
Adjustments to plan based on milestones beaten, met or missed	Can be useful when adjusting the construction plans and processes

Managing Construction : Construction Quality

The measures and measurements which help in understanding the quality would be in terms of "Productivity & Progress with respect to the plans" and "Code quality"

1. Development Progress and productivity metrics (Proceeding as Planned)

   Measures for this would be number of active days spent, assignment scope completed, productivity, efficiency, code churn etc.

   Metrics could be in terms of LoC/effort days, LoC generated/purged etc.

2. How easy is the system to debug, troubleshoot, maintain, integrate and, extend with new functionality (Technical Quality)

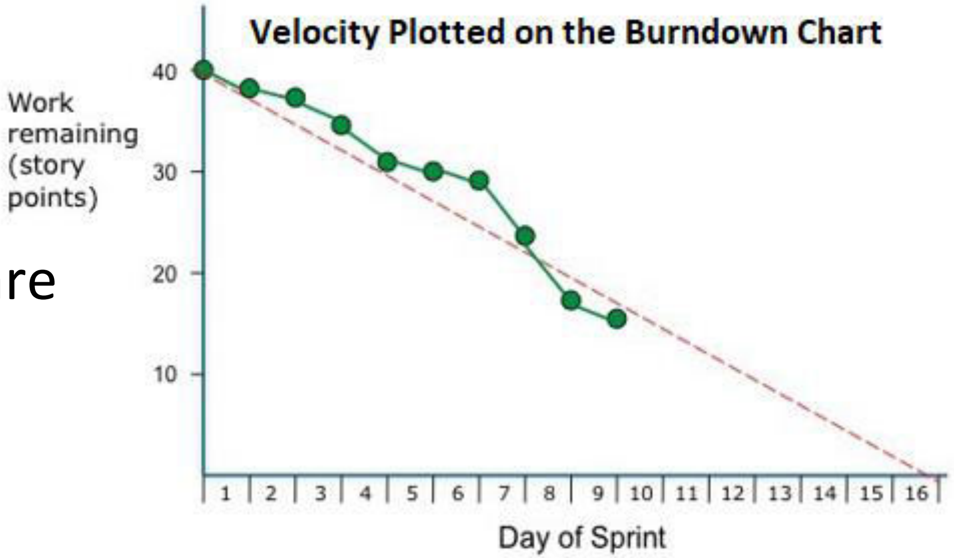
   Measures for this would be obtained from static code analysis which would include Lines of Code (LOC), code complexity, Instruction Path Length etc.

   Metrics would be in terms of No of review errors/KLoC (1000 LoC) etc.

Measurements should typically be deployed to only answer business questions.

Managing Construction : Quality for an Agile Scrum Project would be ..

1. **Sprint Burndown** as seen graphically is a key metric communicating the completion of work throughout the sprint based on story points.
   The goal of the team is to consistently deliver all work, according to the forecast



Burndown chart answers the question, "Are we on track to successfully deliver this sprint's output?

2. **Team Velocity metric** accounts for the "amount" of software or stories the team completes during a sprint. It can be measured in story points or hours, and you can use this metric for estimation and planning.
3. **Throughput** indicates the total value-added work output by the team. It is typically represented by the units of work (tickets) the team has completed within a set period of time.
4. **Cycle Time** stands for the total time that elapses from the moment when the work is started on an item (e.g., ticket, bug, task) until its completion.

… Escaped Defects, True test coverage

Managing Construction : Construction Technical Quality

Activities for ensuring construction quality include:

- **Peer review**: formal name for a common and informal process where the developer shows the code to other developers, looking for advice or comment
- **Unit testing**: The process of writing code (in a framework or test harness) that will call the methods or functions in the code module under test and check the results
- **Test-first**: The practice of designing and building the test harness before writing the code that will be tested. Balances developers tendency to write tests to match their code
- **Code stepping**: Code being tested or reviewed is executed one statement at a time, and the debugger shows the values of variables & the flow of control as each statement executes

Managing Construction : Construction Quality

Activities for ensuring construction quality include: (Contd.)

Pair-programming

Two developers work on same code, with one focusing on function logic and another on syntax and accuracy

Debugging

The process of analyzing code to locate a known defect

Code Inspections

A formal process of peer review. Developer presents code for review to code inspectors

Static analysis

Any process or tool that examines the code without executing it. E.g., peer review and code Inspections, static analysis tools

Code Review

Code Review, or Peer Code Review, is the act of consciously and systematically convening with one's fellow programmers to check each other's code for mistakes

- Two heads are much better than one!
- What to review for?
  - Correctness
  - Error handling
  - Readability
  - Coding standards / Coding guidelines
  - Optimization
- Use Review checklists

Code Inspection

- Software Inspection involves people examining the source representation with the aim of discovering anomalies and defects
- Inspections can check conformance with a specification but not conformance with the customer’s real requirements
- It’s a formalized approach to code/document reviews or peer reviews
- Preconditions
  - A precise specification must be available.
  - Team members must be familiar with the organization standards.
  - Syntactically correct code or other system representations must be available.
  - An error checklist should be prepared.

Code Inspection

Inspection Procedure

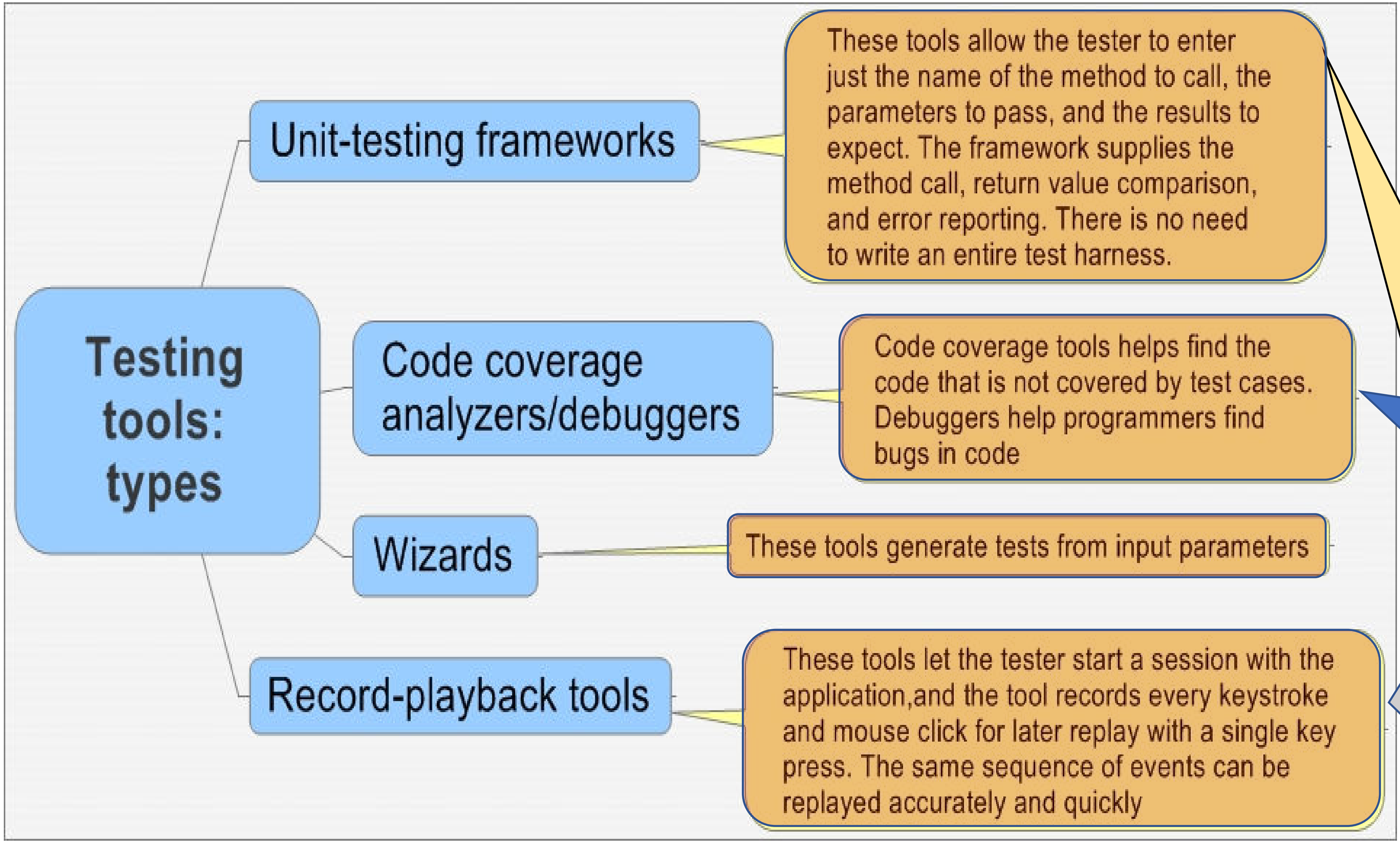
- Inspection Planning – selecting team, location and having the code/documents
- System overview presented to inspection team as part of preparation
- Code and associated documents are distributed to inspection team in advance.
- Individuals prepare where every inspector will inspect the item
- On the specified Inspection logging meeting, Inspection takes place & discovered errors are noted.
- Owners identify modifications necessary, and make the modifications to repair errors discovered.
- Re-inspection may or may not be required post the changes
- All of these are documented and archived

Inspection Checklist

- Checklist of common errors should be used to drive the inspection.
- Error checklists are programming/language dependent & reflect the characteristic errors that are likely to arise in the language.
- In general, the 'weaker' the type checking, the larger the checklist.
  Examples: Initialization, Constant naming, loop termination, array bounds, etc.

SOFTWARE IMPLEMENTATION

Unit Testing Tools



NUnit is a unit testing
fra
pl

- There are number of tools like CoCo

Eg. Salenium

- Primarily it is for automating web applications for testing purposes, but is certainly not limited to just that.

- Boring web-based administration tasks can (and should) also be automated as well

Requirement Traceability Matrix

Sl. No	Req Id	Brief Desc	Architecture Ref Section	Design Ref	Code File Ref	Unit Test Cases	Function/ System test cases
	Pre Filled by earlier Phase						To be Filled in later

THANK YOU

Prof. Phalachandra H.L.
Department of Computer Science and Engineering
phalachandra@pes.edu

SOFTWARE ENGINEERING

SOFTWARE CONFIGURATION MANAGEMENT

Prof. Phalachandra H. L
Department of Computer Science and Engineering

Acknowledgements: Significant portions of the information in the slide sets presented through the course in the class, are extracted from the prescribed text books, information from the Internet and supplemented by my experience. Since these are only intended for presentation for teaching within PESU, there was no explicit permission solicited. We would like to sincerely thank and acknowledge that the credit/rights remain with the original authors/creators only

SOFTWARE CONFIGURATION MANAGEMENT

INTRODUCTION TO SCM

Prof. Phalachandra H. L
Department of Computer Science and Engineering

Context

- All the software (code) constructed as part of Implementation will need to be reliably and effectively converted to executable code (Build), which can be tested and then released to the Customers.
- Given that software engineering is for large projects, with a large number of people working concurrently on generating documentation and code for the product, located at different physical (even geographical) locations, there is a need for approaches or engineering processes which will support
  - Administering or managing the Code line, Branching, Merging or Integration and Version control of the source code
  - Building the Integrated or Merged code or software
  - Packaging the built software
  - Change Management

Generically activities involving all of the above activities (and few more discussed later) are typically considered for discussion as part of software configuration management

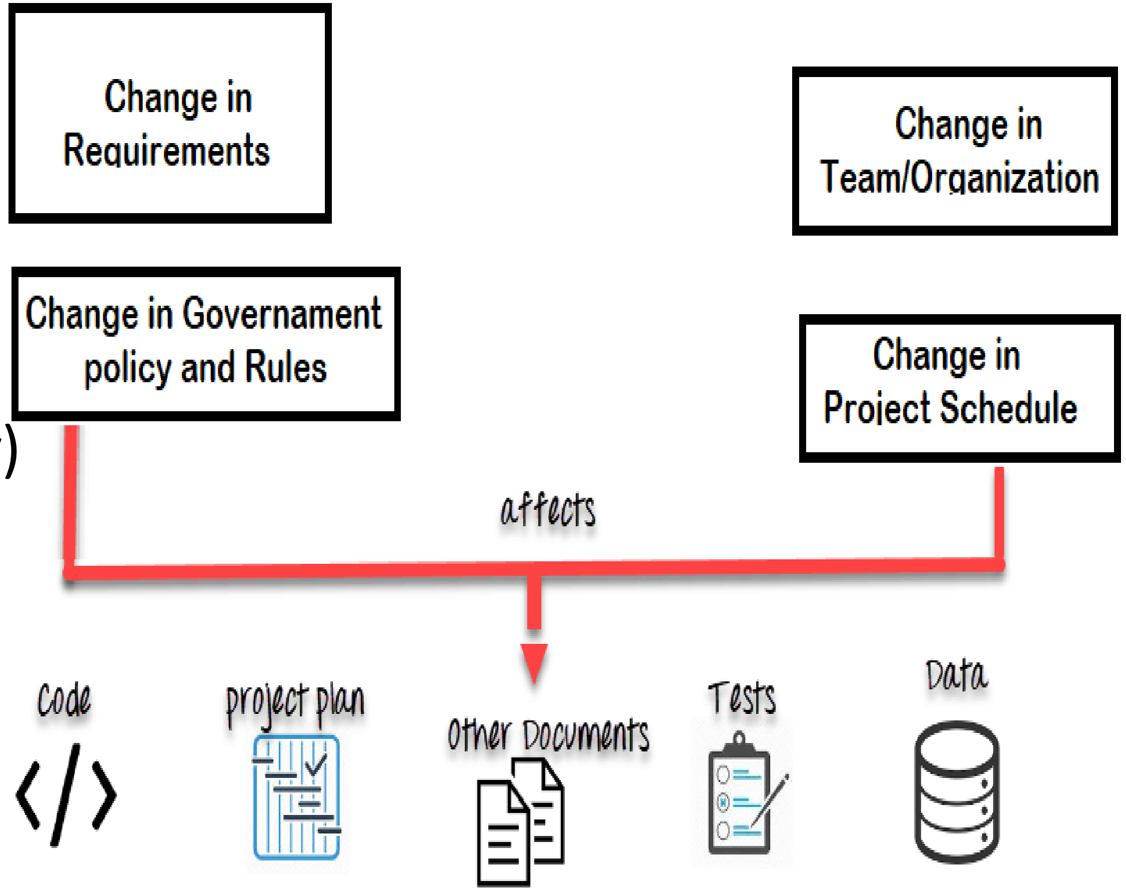
What is Software Configuration Management

- Software Configuration Management is a process to systematically organize, manage and control the changes in the documents, code and other entities which are outcomes of different phases of the Software Engineering process for the software product under consideration.
- The primary goal is to increase productivity by increased and planned coordination among the programmers in a team and eliminates confusion and mistakes
- SCM involves identifying individual elements & configurations, tracking changes, version selection, control and baselining.
- This helps in avoiding configuration related problems, effective management of simultaneous updating of source files, build management, defect tracking

SOFTWARE CONFIGURATION MANAGEMENT
Why SCM

Software engineering in large products/projects with multiple people potentially across different locations need to be reliably managed for

- multiple people concurrently working on the same piece of software & continuously updating the same
- work on more than one version of the software
- working on released systems
- changes in configuration items due to changes to user requirements, budget, schedule etc.
- custom configured systems (different functionality)
- Code/software which must run on different machines and OS's
- Coordination among stakeholders
- Controlling the costs involved in making changes to a system



SCM in Scrum-Agile Approach

SCM is the responsibility of the 'whole team' and will need to be automated as much as possible and educated to the Scrum practitioners.

- Typically, the definitive versions of components are held in a shared project repository.
- Developers copy them into their own workspace, make changes to the code and then use system-building tools to create a new system on their own computer for testing.
- Once they are happy with the changes made, they return the modified components to the project repository.

This makes the modified components available to other team members

Benefits of SCM

The effective use of an SCM system also:

- Permits the orderly development of software configuration items.
- Ensures the orderly release and implementation of new or revised software products.
- Ensures that only approved changes to both new and existing software products are implemented and deployed.
- Ensures that the software changes that are implemented are in accordance with approved specifications.
- Ensures that the documentation accurately reflects updates.
- Evaluates and communicates the impact of changes.
- Prevents unauthorized changes from being made.

Configuration Management Roles

Configuration Manager

- Responsible for identifying configuration items.
- Defining the procedures for creating and promotions and releases

Change Control Board (CCB) member

- Responsible for approving or rejecting change requests

Developer

- Creates versions triggered by change requests or the normal activities of development
- Checks in changes & resolves conflicts

Auditor

- Responsible for validating the processes followed for selection and evaluation of promotions for release
- Ensures the consistency and completeness of the release

Software Configuration Management Planning

1. Software Configuration Planning is done by a configuration Manager who would do this as a dedicated or part time role
2. Software configuration management planning starts during the early phases of a project.
3. The outcome of the SCM planning phase is the Software Configuration Management Plan (SCMP)
4. The SCMP can either follow a public standard like the IEEE 828, or an internal (e.g. company specific) standard.

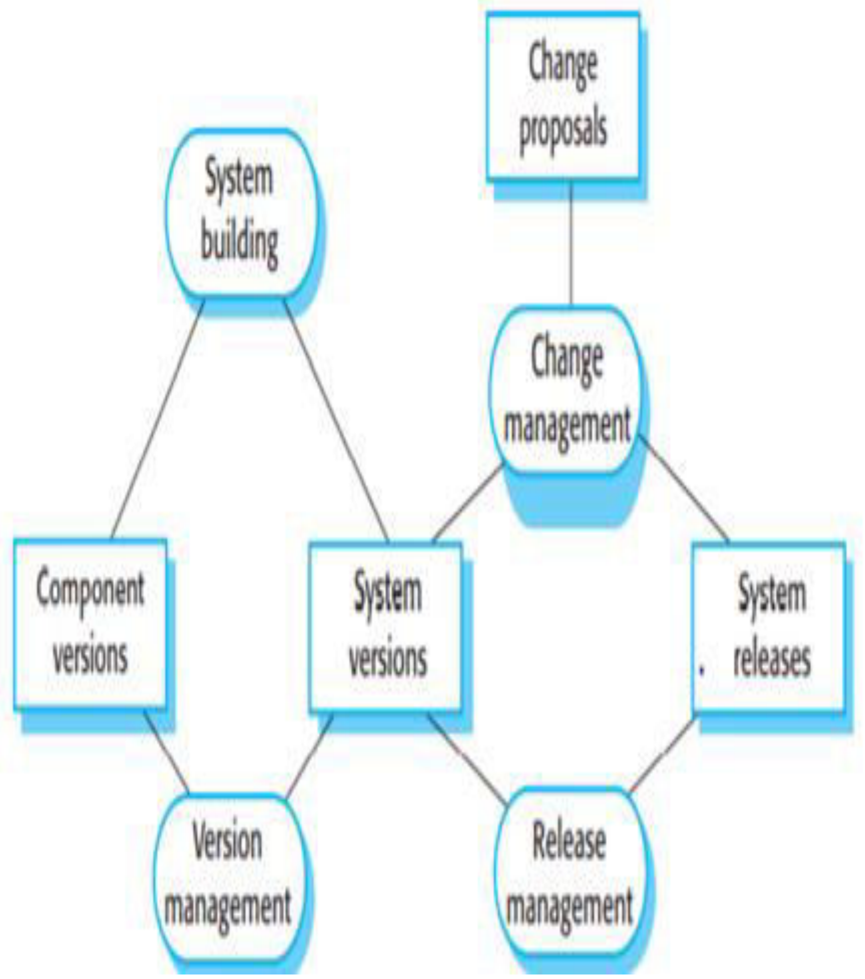
Software Configuration Management Plan

- Defines the *CIs* which need to be managed and a naming scheme.
- Defines *who takes responsibility* for the CM procedures and the creation of baselines.
- Defines ***policies*** *for* ***change control*** and **version management**.
- Describes the *tools* which should be used to assist the CM process and any limitations on their use.
- Defines the *configuration management database* used to record configuration information.

SOFTWARE CONFIGURATION MANAGEMENT

Configuration Management Activities

1. Configuration item identification
2. Configuration Management Directories
3. Baselining
4. Branch management
5. Version Management
6. Build Management
7. Install
8. Change Management
9. Promotion management
10. Release management
11. Defect Management



THANK YOU

Prof. Phalachandra H.L.
Department of Computer Science and Engineering
phalachandra@pes.edu

SOFTWARE ENGINEERING

SOFTWARE CONFIGURATION MANAGEMENT

Prof. Phalachandra H. L
Department of Computer Science and Engineering

Acknowledgements: Significant portions of the information in the slide sets presented through the course in the class, are extracted from the prescribed text books, information from the Internet and supplemented by my experience. Since these are only intended for presentation for teaching within PESU, there was no explicit permission solicited. We would like to sincerely thank and acknowledge that the credit/rights remain with the original authors/creators only

Configuration Management Activities (Recap)

1. Configuration item identification
2. Configuration Management Directories
3. Baselining
4. Branch management
5. Version Management
6. Build Management
7. Install Management
8. Change Management
9. Promotion management
10. Release management
11. Defect Management

SOFTWARE CONFIGURATION MANAGEMENT

Some of the basic activities and structures of SCM

Dr. Phalachandra H. L
Department of Computer Science and Engineering

1. Configuration Items

Definition : Any independent or aggregation of hardware, software, or both, that is designated for configuration management and treated as a single entity in the configuration management process"

- Software configuration items could be
  - all type of code files
  - drivers for tests
  - Requirement, analysis, design, test & other non-temporary documents
  - user or developer manuals
  - system configurations (e.g. version of compiler used) .. Etc.
- Systems can have software and hardware configuration items (CPUs ..)
- Large projects typically produce thousands of entities (files, documents, data ...) which are uniquely identified and can potentially be brought under configuration management control

1. Configuration Items (Cont.)

Challenges associated with configuration Items :

Since not all entities in an project needs to be under configuration management, this leads to two challenges

1. What are those items which need to be in Configuration Control (Selection)

Some items must be maintained for the lifetime of the software.

- Selecting the right configuration items is a skill that takes practice
  - Very similar to object modeling
  - Use techniques similar to object modeling for finding CIs!
    - Find the CIs
    - Find relationships between CIs drivers for tests

2. When do you start to place entities under configuration control?

Considerations on when to start:

- Starting with CIs too early introduces too much bureaucracy
- Starting with CIs too late introduces chaos

2. SCM Directories - Code as an CI

Programmer's Directory (IEEE: Dynamic Library)

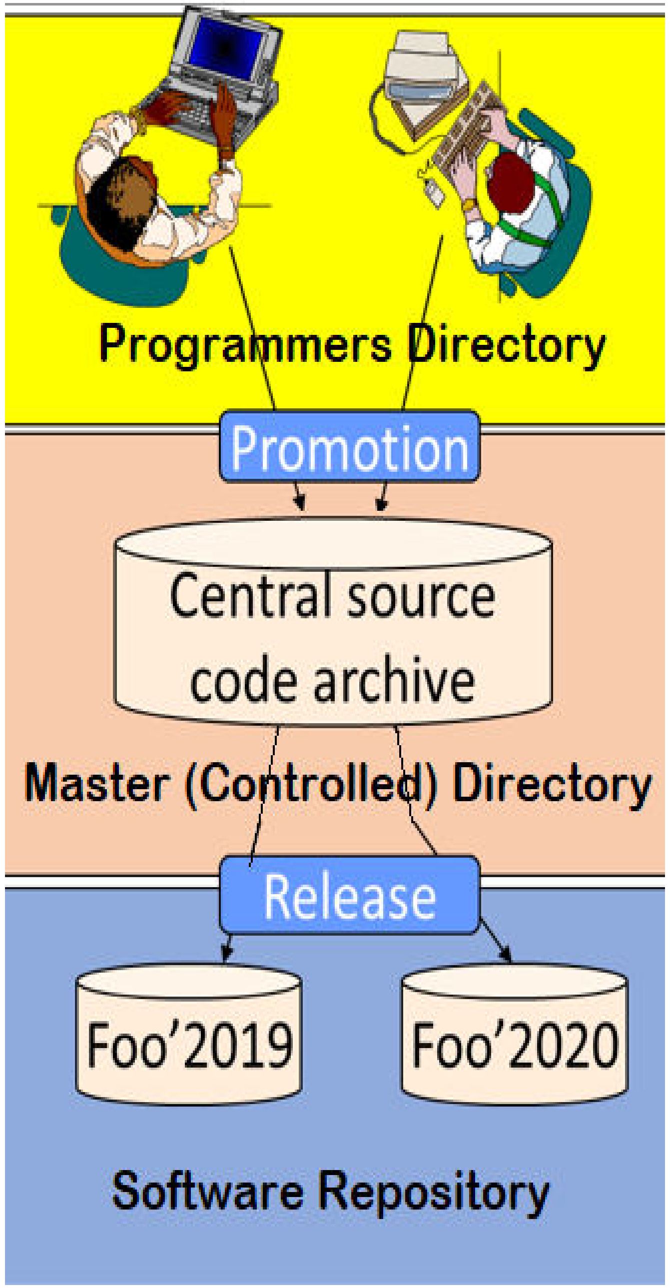
- Library for holding newly created or modified software entities.
- The programmer's workspace is controlled by the programmer only.

Master Directory (IEEE: Controlled Library)

- Manages the current baseline(s) and for controlling changes made to them.
- Entry is controlled, usually after verification.
- Changes must be authorized.

Software Repository (IEEE: Static Library)

- Archive for the various baselines released for general use.
- Copies of these baselines may be made available to requesting organizations.



3. Baselines

Definition : A specification or product that has been formally reviewed and agreed to by responsible management, that thereafter serves as the basis for further development, and can only be changed through formal change control procedures

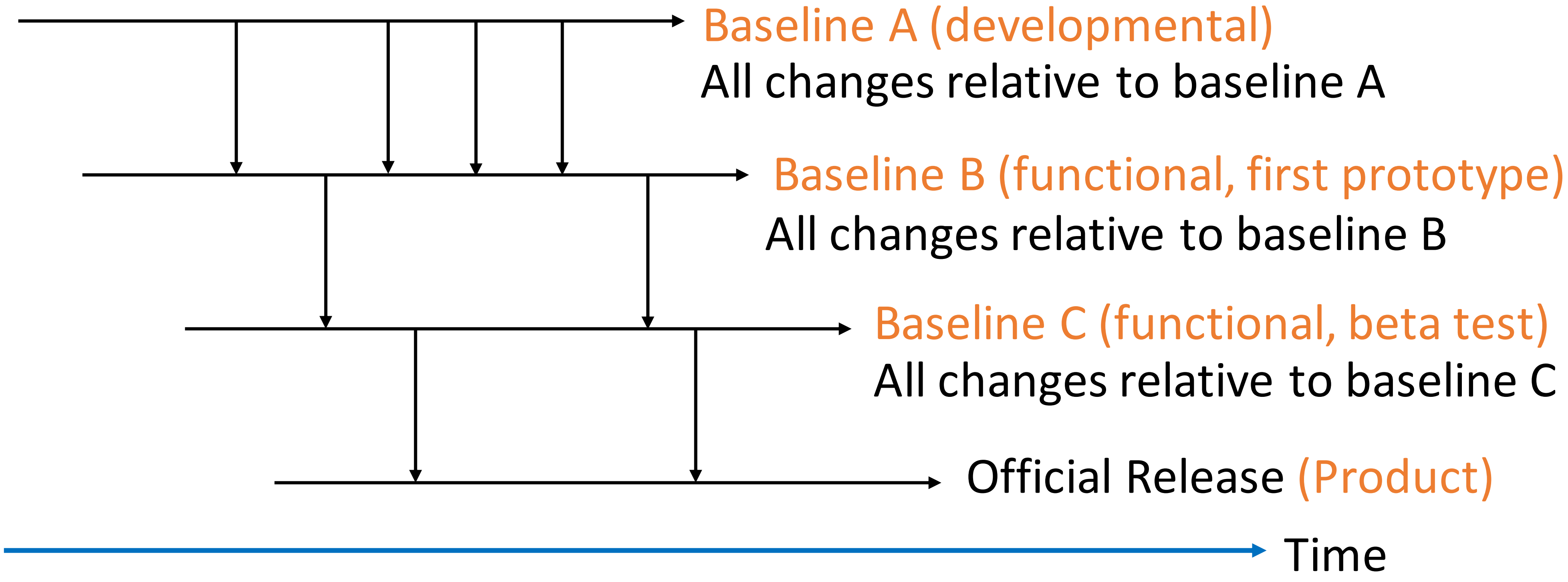
As systems are developed, a series of baselines are developed, usually after a review (analysis review, design review, code review, system testing, client acceptance, ...)

Examples:

Baseline A: All the API have completely been defined; the bodies of the methods are empty.

Baseline B: All data access methods are implemented and tested.

3. Baselines (Cont.)



Are baselines tied to project milestones?

Many are, but is not mandatory

There are many naming schemes for baselines (1.0, 6.01a, ...)

THANK YOU

Prof. Phalachandra H.L.
Department of Computer Science and Engineering
phalachandra@pes.edu

SOFTWARE ENGINEERING

SOFTWARE CONFIGURATION MANAGEMENT

Prof. Phalachandra H. L
Department of Computer Science and Engineering

Acknowledgements: Significant portions of the information in the slide sets presented through the course in the class, are extracted from the prescribed text books, information from the Internet and supplemented by my experience. Since these are only intended for presentation for teaching within PESU, there was no explicit permission solicited. We would like to sincerely thank and acknowledge that the credit/rights remain with the original authors/creators only

SOFTWARE CONFIGURATION MANAGEMENT

Branch, Version, Build and Install Management

Dr. Phalachandra H. L
Department of Computer Science and Engineering

SOFTWARE CONFIGURATION MANAGEMENT

Configuration Management Activities (Recap)

1. Configuration item identification
2. Configuration Management Directories
3. Baselining
4. Branch management
5. Version Management
6. Build Management
7. Install Management
8. Change Management
9. Promotion management
10. Release management
11. Defect Management

4. Branch Management

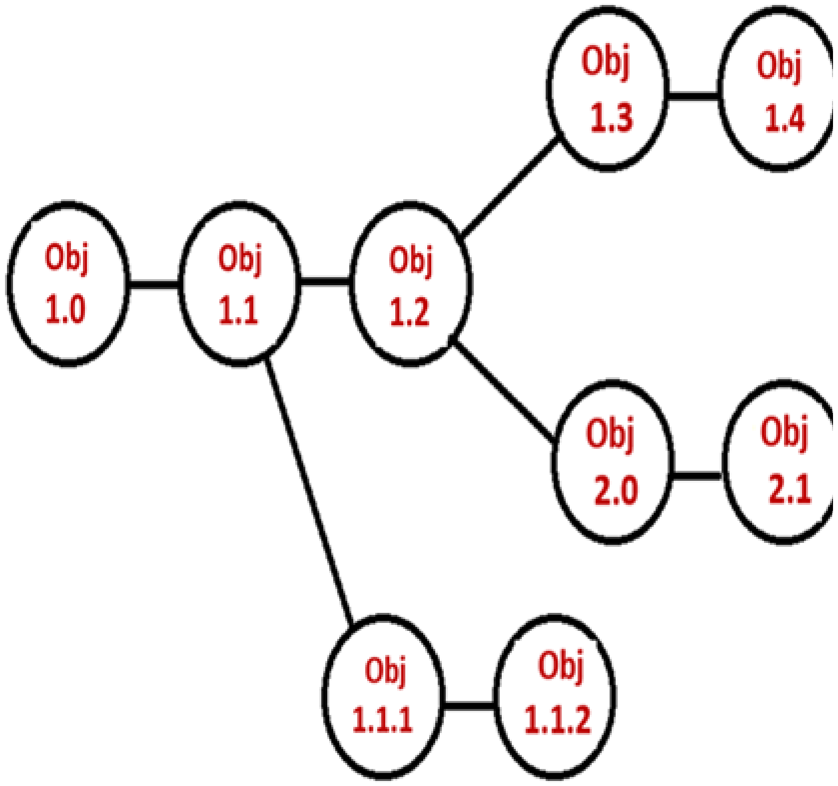
- A codeline is a progression of the set of source files, and other artifacts, which make up software components as they change over time
- A branch is a copy or clone of all, or at least a portion of the source code (or a code line) within the repository.
- Code branching is done to reasons like
  - supporting concurrent development
  - capturing of solution configurations
  - supporting multiple versions of a solution
  - to enable developers to experiment in isolation enabling working without impact to others in parallel.
- Branching helps in his keeping the overall product stable in situations as above.
- Merging is bringing back and integrating the changes done in branches so all users of the branch can see.
- Frequent merging from the related branched code lines into the working branch helps decreasing the likelihood and complexity of a merge conflict

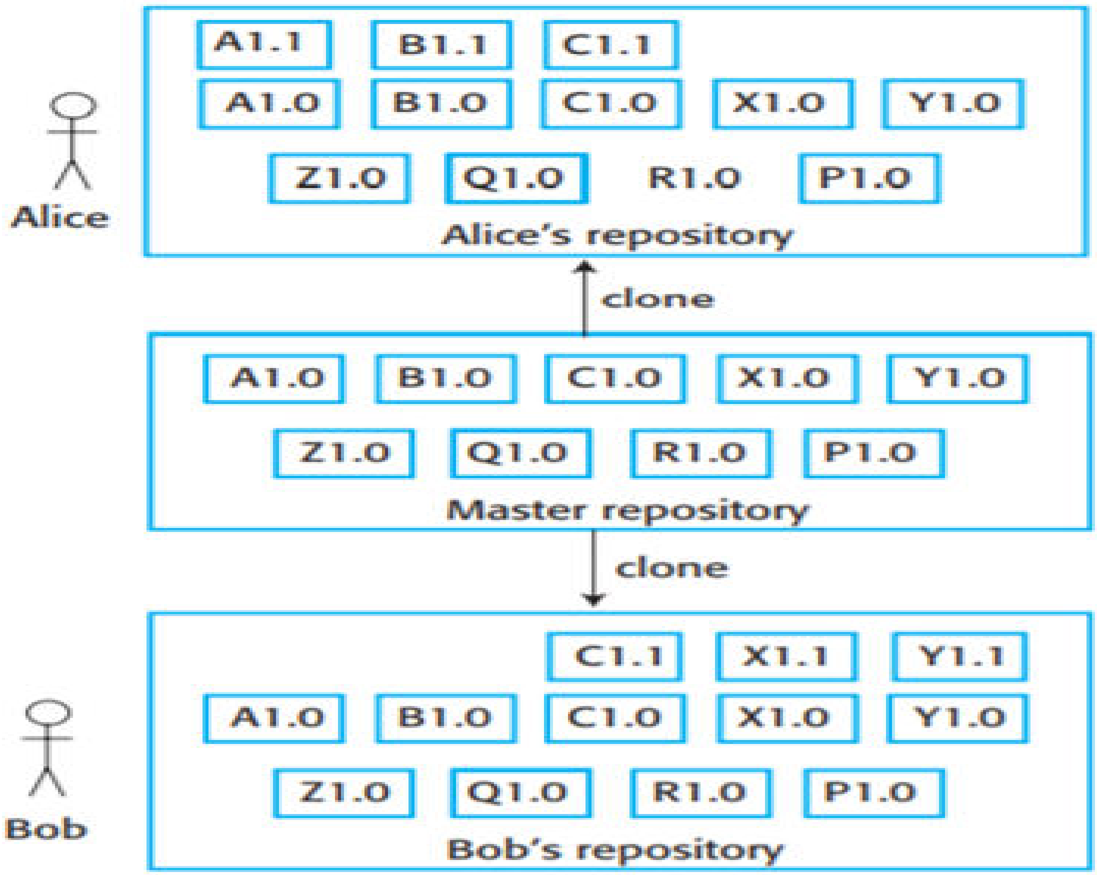
4. Branch Management

- There can be many branching strategies available which may be applied in combination.
  - Single branch (trunk based)
  - Branch by customer/organization
  - Branch by developer/workspace
  - Branch by module/component
  - …..
- Branching management entails having a well defined branching policy, having an owner for the code lines and use branches for release

4. Branch and Version Management

The following examples showcases some of branching and the process of keeping track of different versions of software components and the systems in which these components are used

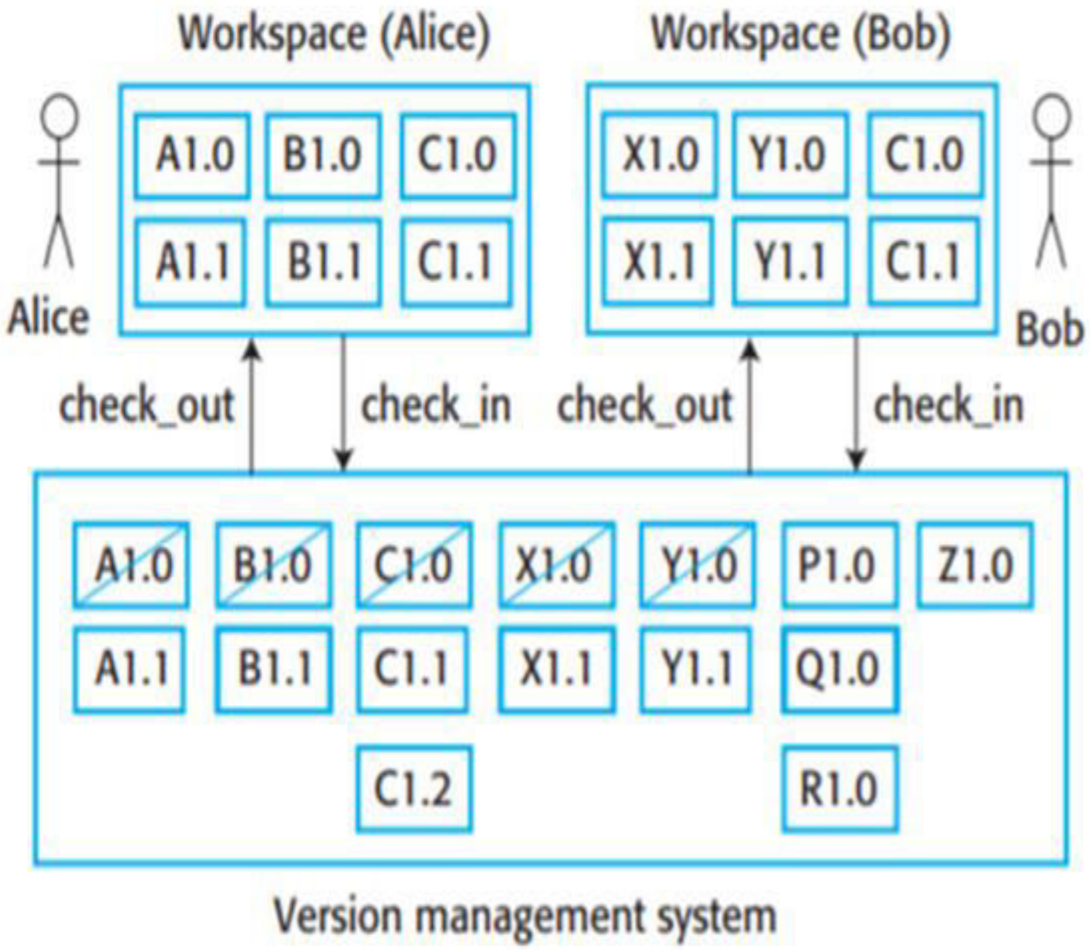




5. Version Management

Version management is the process of keeping track of different versions of software components and the systems in which these components are used. Typically tools like Git are used for version management of source code

- Changes to a version are usually identified by a number, termed the "revision number"



5. Key Features of Version control System

- All changes are attributable or traceable
- Change history is recorded so every thing can be tracked and reverting back if needed is easier
- Better conflict resolution
- Easier code maintenance and code quality monitoring
- Less software regression
- Better organization and a better communication

6. Build Management - 1

- Build management is actually a process of creating the application program for a software release by taking all the relevant source code files and libraries and compiling and linking them to create build artefacts, such as binaries or executable program etc.
- This could involve collecting all the components making up an product component which can be released and performing all the tasks of compile, link and sanity test the binary.
- Build is typically done using tools like the basic Make, Apache Ant, MS Build, Maven etc. which help in building software components

6. Build Management - 2

- The process of build involves compilation of the various files, in the correct order and linking all the libraries in the appropriate order.
- If the source code in a particular file has not changed then it may not need to be recompiled (may not rather than need not because it may itself depend on other files that have changed).
- Sophisticated build utilities and linkers attempt to refrain from recompiling code that does not need it, to shorten the time required to complete the build.
- Build automation involves scripting the process

6. Build Management - 3

1. Build Process

This consists of

- Fetching the code from the source control repository
- Compiling the code and checking the dependencies/modules
- Linking the libraries, code, files, etc. accordingly.
- Running the automated/manual unit tests
- Once successfully passed, Build the artefacts and store them.
- Archive the build logs.
- Send the notification emails.
- Typically this may lead to a version number change

6. Build Management - 4

1. **Types of Builds:**
   - Full Build
   - Incremental Build – Optimized Build

3. **Build Triggers:**
   - Manual Build Trigger
   - Scheduled Build Trigger
   - Source code repository Build Trigger
   - Post Process Build Trigger

7. Install Management - 1

- Software installation is the first interaction developers have with their new user.
- Its impactful in terms of being the brief period of opportunity and if the installation fails, it is likely that your user, short of attention-span and patience, will set a negative perception
- This process could involve placing multiple files containing executable code, downloading or copying from a repository or drives more detailed executable files, images, libraries, configuration files developed by different developers, from the internet.

7. Install Management - 2

- Software installation can also involve interaction with OS functions for validating the resources needed, permissions, versions, identifiers to ensure enforcement of licences and registration.
- This may also involve customizations for localizations
- Given that Installation involves complex detailed steps, this leads to the need automation and usage of tools like Zip/tar/Shell Scripts in a trivial state to InstallAware, InstallShield, Jenkins, Vagrant

THANK YOU

Prof. Phalachandra H.L.
Department of Computer Science and Engineering
phalachandra@pes.edu

SOFTWARE ENGINEERING

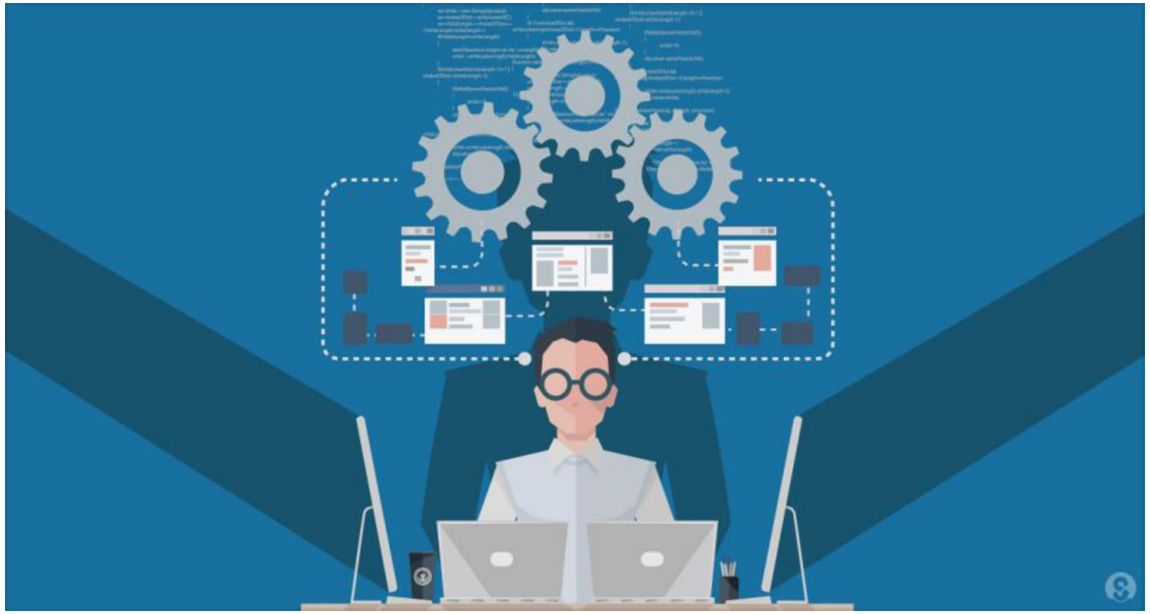
SOFTWARE CONFIGURATION MANAGEMENT

Dr. Phalachandra H. L
Department of Computer Science and Engineering

Acknowledgements: Significant portions of the information in the slide sets presented through the course in the class, are extracted from the prescribed text books, information from the Internet and supplemented by my experience. Since these are only intended for presentation for teaching within PESU, there was no explicit permission solicited. We would like to sincerely thank and acknowledge that the credit/rights remain with the original authors/creators only

SOFTWARE CONFIGURATION MANAGEMENT

Change, Promotion, Release & Defect Management



Dr. Phalachandra H. L
Department of Computer Science and Engineering

SOFTWARE CONFIGURATION MANAGEMENT

Configuration Management Activities (Recap)

1. Configuration item identification
2. Configuration Management Directories
3. Baselining
4. Branch management
5. Version Management
6. Build Management
7. Install Management
8. Change Management
9. Promotion management
10. Release management
11. Defect Management

8. Change Management

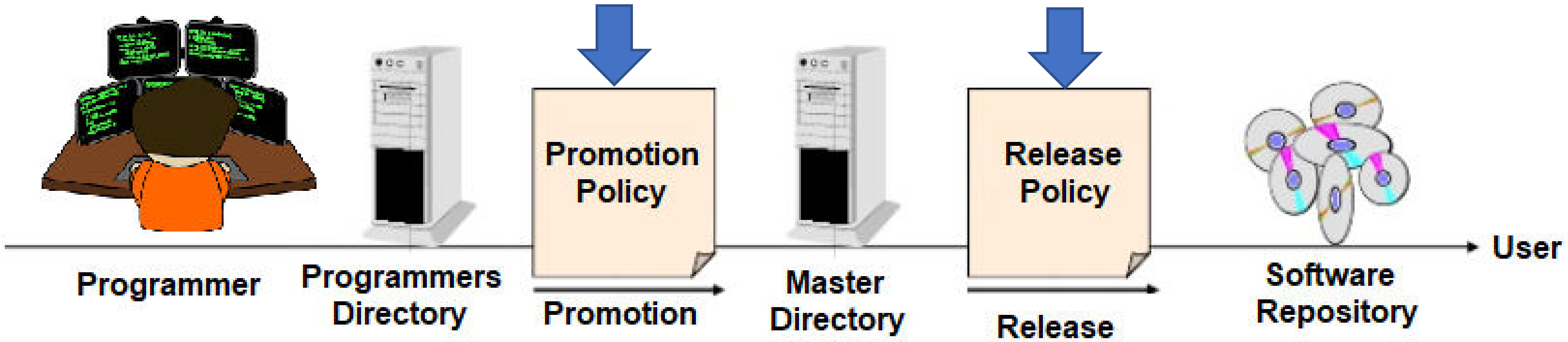
- A change could result in the creation of a different version or a release of the software
- Deals with the changes to the Configuration Items (CIs) which have been baselined
- General change process
  - Change is requested (anytime by anyone)
  - A Unique identification is associated with this change being requested (Change Id, Requestor, Date etc.) and typically logged into tools
  - The change is assessed for impact to other modules, branches and categorized
  - Decision on the change - Accepted or Rejected
  - All of these activities are mostly tool driven
  - If accepted, the change is implemented and validated
  - Plans done and executed for documentation, versioning, merging and delivery
  - The implemented change is also audited.

8. Change Management

- The complexity of the change management process varies with the project
  - Small projects perform change requests informally (and fast)
  - Complex projects require detailed change request forms and the official approval by one or more managers or through Configuration Control Board (CCB) (which exists in large project).
- Information required to process a change to a baseline would need to include
  - A description of the proposed changes
  - Reasons for making the changes
  - List of other items affected by the changes
- Tools, resources, and training are required to perform baseline change assessment
  - File comparison tools to identify changes
  - Other Resources and training depending on size and complexity of project

Controlling Changes :

Changes to code are typically controlled at two points

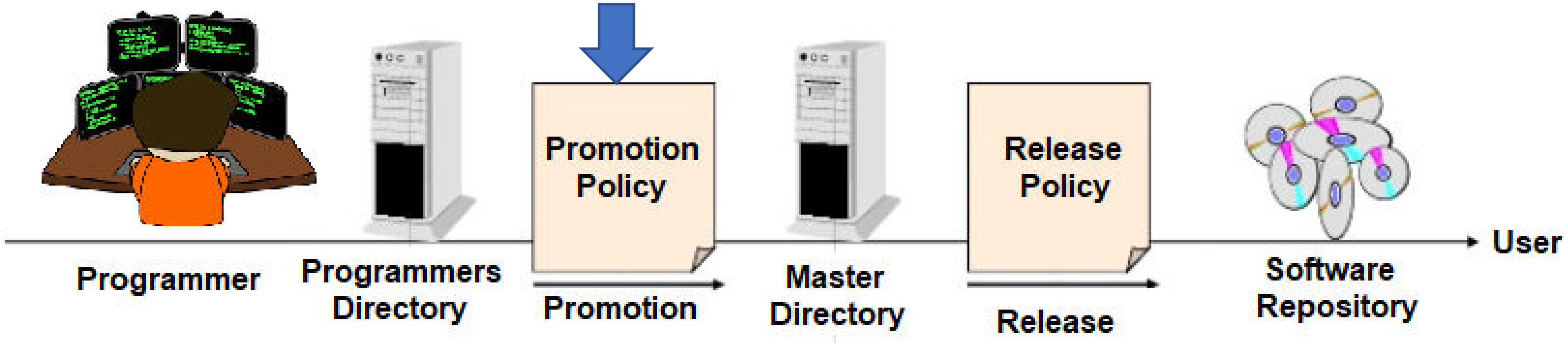


Change policies : These promotion and release change policies are approaches for controlling change. These could be

- Informal (good for research type environments and promotions)
- Formal approach (good for externally developed CIs and for releases)

The purpose of change policies is to guarantee that each version, revision or release conforms to commonly accepted criteria and a consistent process is applied to how things are done. These change policies are enforced through engineering processes and tools. They are also Audited to ensure they are conformant to the processes defined.

9. Promotion Management

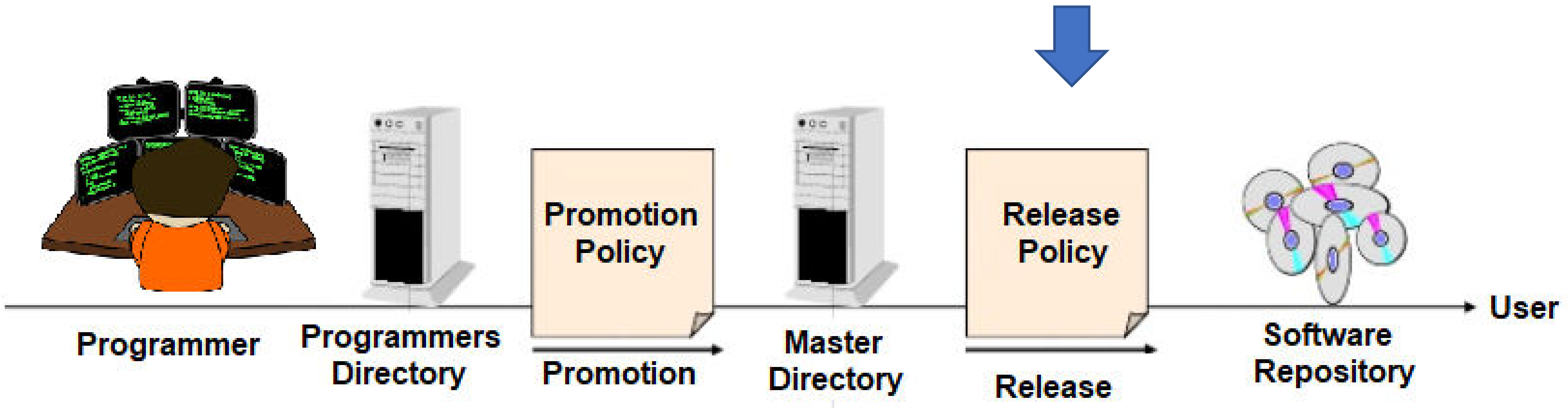


Changes made by a programmer is only available within the programmers environment and hence will need be promoted to a central master directory based on certain promotion policies.

This promotion could be based on the baselining criteria which was planned for, and would include some kind of verification of meeting the criteria. Post that it would be authorized to be promoted and moved to the Master directory, which would then be available to other dependent modules

E.g. “No developer is allowed to promote source code which cannot be compiled without errors and warnings.”

10. Release Policy and Management



The code in the Master directory is then released to customers and moved to the Software repository. This movement is done based on the release policies.

The release policy would be based on gating quality criteria which was planned for, and would include some kind of verification of the metrics to be meeting the criteria, post which it would be authorized to be released to the customers and archived in the software repository.

E.g. for the release policy

"No baseline can be released without having been beta-tested by at least 500 external persons."

10. Release Management

- Release management is an overloaded terminology. It is considered as a process of managing, planning, scheduling and controlling a software build through different stages and environments; including testing and deploying software releases.
- Release management could also be looked at as making a software component to be available to a customer by deploying a system release (which is a version of a software system) to customers. Since we are looking at each of the others (Build, Install etc) independently, we are looking at the release management as deployment.
  - Complete Product (Major or Minor)
  - Components of Product

10. Release Management : Patches

- A patch is a set of changes to a computer program or its supporting data designed to update, fix, or improve it.
- This includes fixing security vulnerabilities and fixing other bugs (for bug fixes.)
- Patches are often written to improve the functionality, usability, or performance of a program.
  - Generic Patch
  - These also could be binary patches, source code patches
  - Hot fix
  - Emergency or Site Specific Patch
  - Unofficial patch
  - Patch Bundle
  - Service pack - A service pack or SP or a feature pack (FP) comprises a collection of updates, fixes, or enhancements to a software program delivered in the form of a single installable package

Version Vs Revision Vs Release

Release:

- The formal distribution of an approved version.

Version:

- An initial release or re-release of a configuration item associated with a complete compilation or recompilation of the item. Different versions can have different functionality.

Revision:

- Change to a version that corrects only errors in the design/code, but does not affect the documented functionality.

The usage of the terminology presented here is not strict but varies for different configuration management systems.

A 3 digit scheme is quite common:

7.5.5

Release (Customer)

Version (Developer)

Revision (Developer)

11. Bug or Defect Management - 1

- Bug is an consequence of a coding fault
- Defect is a variation or deviation from the expected business requirements to have been implemented
- Approaches to manage these would be similar and involve following the Defect Management Process & Generation and considerations of Metrics
- All of this bug/defect management is often driven through the defect management steps in usage of tools like Bugzilla

11. Bug or Defect Management - 2

- Defect Management Process would contain the following steps

1. Discover
2. Reporting, logging into the tool (if being used) with an unique identifier
3. Validation
4. Analysis and Categorization (Critical, High, Medium and Low)
5. Request and Approval for fix or change
6. Resolution (Assignment, Schedule, Fix, test and report on resolution)
7. Verification by submitter
8. Merging of the code
9. Updating of the version number
10. Plan for release of the fix to the customer
11. Closure
12. Reporting

THANK YOU

Prof. Phalachandra H.L.
Department of Computer Science and Engineering
phalachandra@pes.edu

SOFTWARE ENGINEERING

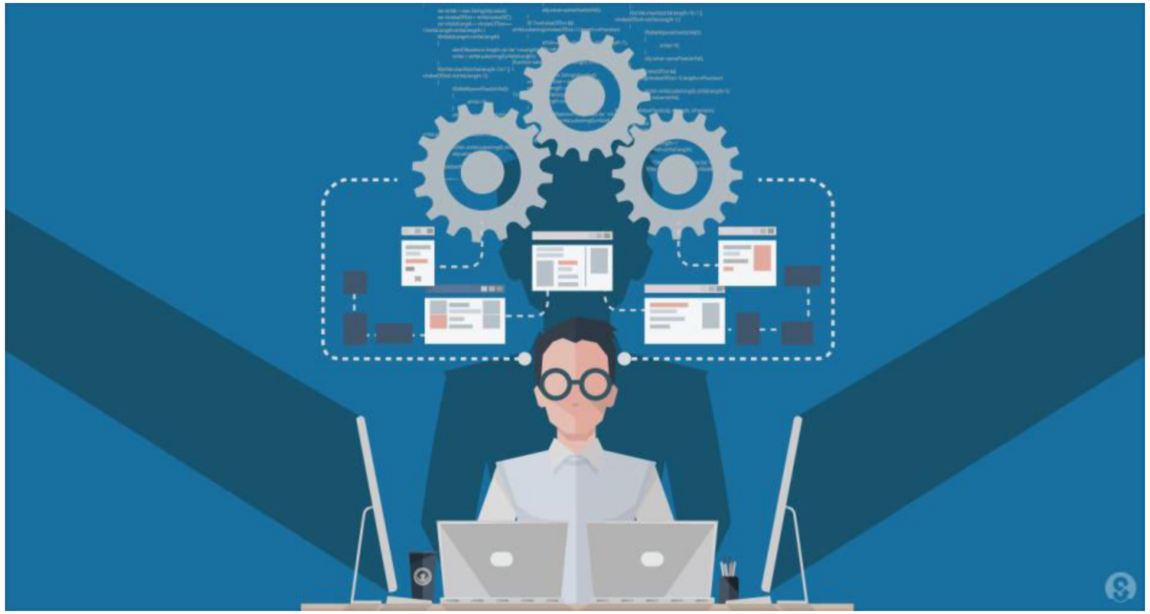
SOFTWARE CONFIGURATION MANAGEMENT

Dr. Phalachandra H. L
Department of Computer Science and Engineering

Acknowledgements: Significant portions of the information in the slide sets presented through the course in the class, are extracted from the prescribed text books, information from the Internet and supplemented by my experience. Since these are only intended for presentation for teaching within PESU, there was no explicit permission solicited. We would like to sincerely thank and acknowledge that the credit/rights remain with the original authors/creators only

SOFTWARE CONFIGURATION MANAGEMENT

Software Configuration Management Tools



Prof. Phalachandra H. L
Department of Computer Science and Engineering

Software Configuration Management Tools

Software Configuration Management Tools are the tools and utilities used in administering source code, building software, install packaging, defect tracking, change management and managing software configurations.

1. Source Code Administration Tools : These are tools used for source code control.
2. Software Build Tools: Tools which are used for building source code
3. Software Installation Tools: Tools which are used for packaging and installing the products
4. Software Bug Tracking Tools : Tools which are used for tracking bugs and changes associated with them

1. Source Code Administration Tools

- RCS
  - GNU - very old but still in use; only version control system
- CVS (Concurrent Version Control)
  - based on RCS, allows concurrent working without locking http://www.cvshome.org/
  - CVSWeb: Web Frontend to CVS
- ClearCase (Linux, Solaris, Windows)
  - Multiple servers, process modeling, policy check mechanisms http://www.rational.com/products/clearcase/
- GitHub
  - GitHub is a development platform where code is hosted for version control and projects can be managed

GitHub Overview

- Create repository, add contributors. Can access using ssh or https.
- Changes can be pushed as commits.
- Each commit is logged with a commit number and message
- Can have different branches to add code to. Master branch is auto cloned.
- Built on open source model, hence another person can fork a project, make changes and request authors to merge by sending a Pull Request. Authors can check and merge.
- Every merge is a change to original project, a git diff is done and only the changes are logged on

GitHub Overview

$git init   #initialise local git repo

$git status   #shows status, untracked files, commits, etc.

$git add <files>   #brings untracked files to git's notice

$git commit -m "text"   #commits

$git remote add origin http://github.com/shrinidhir/JavaProjects
#inform git about the existence of a remote repo

$git remote -v   #gives list of all the remote origins your local repository knows about

$git push   #push changes onto remote repo

$git push origin master   #push just the master branch of the repo

2. Source Code Build Tools

Software build refers either to the process of converting source code files into standalone software artefact(s) that can be run on a computer, or the result of doing so. One of the most important steps of a software build is the compilation process where source code files are converted into executable code.

- Make

  Make is a utility that automatically builds executable programs and libraries from source code by reading files called makefiles which specify how to derive the target program
- CruiseControl

  CruiseControl is an open source tool for continuous software builds.

Source Code Build Tools (Contd.)

- FinalBuilder

  FinalBuilder is a powerful Automated Build & Release Management tool that simplifies software build automation.
  By packaging an extensive library of pre-written scripts into a graphical IDE, FinalBuilder equips you to easily define and maintain a reliable build process.

- Maven :

  Maven is a software project management and comprehension tool.
  Based on the concept of a project object model (POM), Maven can manage a project's build, reporting and documentation from a central piece of information.

Source Code Build Tools (Contd.) : Makefile

Typical Makefile

```
CC = gcc
CFLAGS = -g
LDFLAGS=

all: helloworld

helloworld: helloworld.o
	# Commands start with TAB not spaces
	$(CC) $(LDFLAGS) -o $@ $^

helloworld.o: helloworld.c
	$(CC) $(CFLAGS) -c -o $@ $<

clean: FRC rm -f helloworld helloworld.o
```

Source Code Build Tools (Contd.) : Maven

- Maven simplifies and standardizes the project build process. It handles compilation, distribution, documentation, team collaboration and other tasks seamlessly. Maven increases reusability and takes care of most of the build related tasks
- Can support multiple development team environments, which are reusable and supports creating of reports, checks, build and testing automation setups
- Contains a standard directory layout and default build lifecycles with environment variables to support it. Using env variables like

source code	${basedir}/src/main/java
Resources	${basedir}/src/main/resources
Tests	${basedir}/src/test
Complied byte code	${basedir}/target
distributable JAR	${basedir}/target/classes

3. Software Installation Tools

- Install tools are cross platform tools that produce installers for windows, MacOS, Linux etc.
- An installation program or installer is a computer program that installs files, such as applications, drivers, or other software, onto a computer
- Installers could be custom installers specifically made to install the files they contain or General purpose installers which work by reading the contents of the software package to be installed.
- Installation could use something called Bootstrapper (small installer which runs first, does the pre-requisites and updates the big bundle for installation)
- Most of these also provide features for rollback, clean-up etc. as features

3. Software Installation Tools (Contd.)

- DeployMaster (Windows)
  - DeployMaster is one of the solutions to distribute Windows software or other computer files, via the Internet or on CD or DVD.
  - DeployMaster works with Windows 95, 98, ME, NT4, 2000, XP, 2003 and Vista.
- InstallAware (Windows)
  - Focusing solely on the Windows Installer platform for Microsoft Windows operating systems, InstallAware supports Internet deployment of software.
- InstallShield
  - InstallShield Installer simplifies the process of creating reliable Windows Installers, MSI packages and InstallScript installers for Window systems.
  - InstallShield is the de facto standard for MSI installations.
- Wise Installer/Wise Installation Studio (Windows)
  - Wise Installation Studio 7.0 allows creation of installations that configure & install Microsoft Windows applications to desktops, servers, mobile devices or the web.

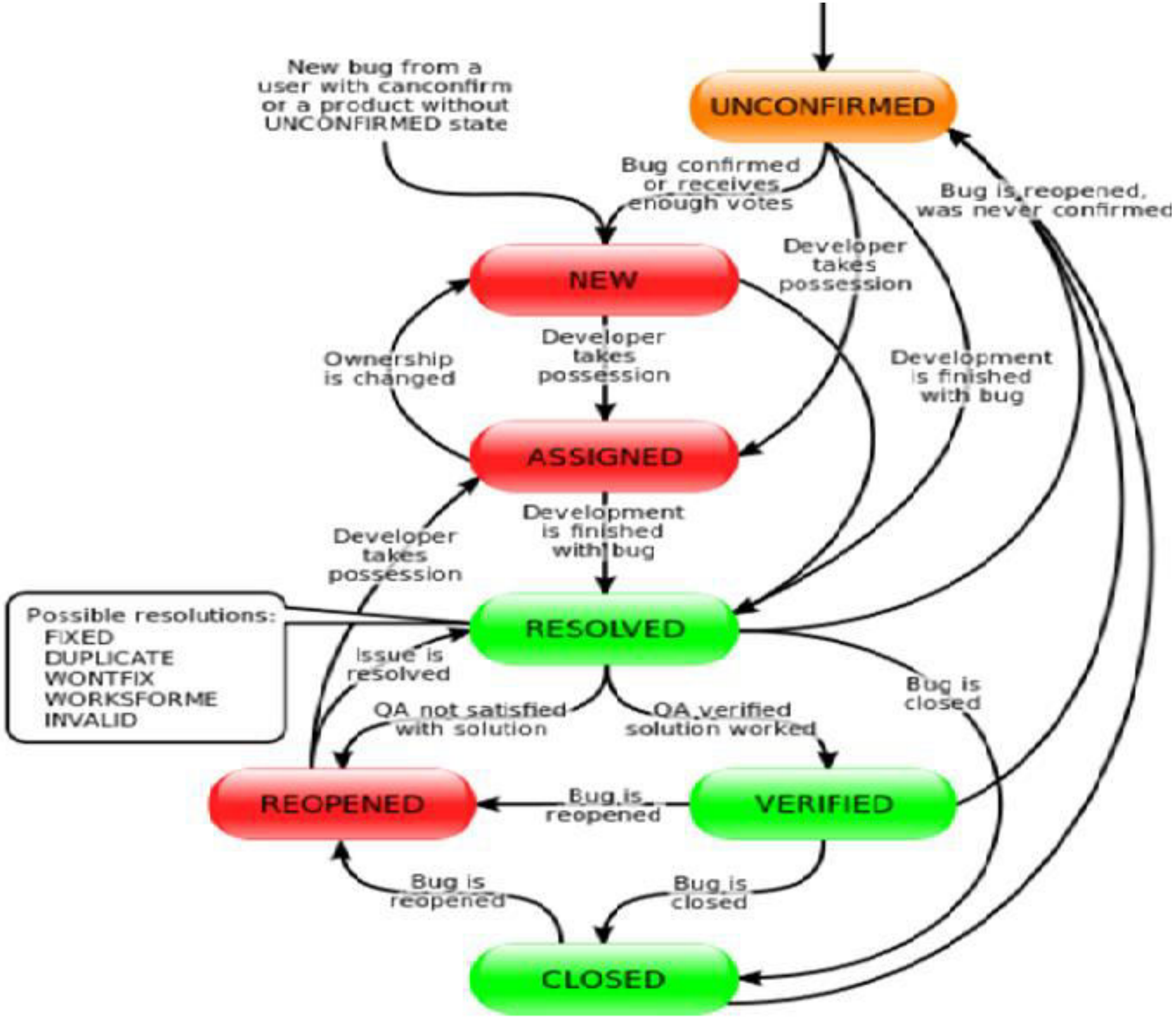
4. Software Bug/Defect Tracking Tools

- A bug tracking system or defect tracking system is a software application that keeps track of reported software bugs in software development projects
- It may be regarded as a type of issue tracking system
- Bug-tracking system is to provide a clear centralized overview of development requests (including both bugs and improvements)
- Typical bug tracking systems will support the life cycle for a bug which is tracking the status of the bug through it being logged to resolution

E.g. Bugzilla, FogBugz, Trac Edgewall Software, Backlog, Mantis
TestTrack Pro, nTask, OpenProject, Remedy Action Request System
from BMC Software, Chart, etc.

SOFTWARE CONFIGURATION MANAGEMENT

4. Software Bug/Defect Tracking Tools : Bugzilla



THANK YOU

Prof. Phalachandra H.L.
Department of Computer Science and Engineering
phalachandra@pes.edu