• Software process models (e.g., waterfall, linear sequential, incremental, agile, etc.)

Classical Process Models

– Waterfall

Requirements Phase

Specification Phase (Analysis)

Planning Phase

Design Phase

Implementation Phase

Integration and Testing

Maintenance

Retirement

– Linear Sequential

Feedback loops to correct uncovered faults

Testing and Verification at each phase

Documentation at each phase

Each phase is completed before next phase can begin

Problems:

Real projects don’t often run sequentially

Customers must have patience

Development is often delayed, i.e., “blocking states”

Specifications may not reflect client expectations

Staffing problems, e.g., “tall, narrow” developers versus “short, wide” developers

– Prototyping Model

Modified Sequential Model

A prototype is constructed to determine system requirements and specifications

Prototype is used as a tool to determine clients needs

Numerous problems can be uncovered during prototype development and evaluation

Problems:

Prototype is viewed by the customer and management as a completed system

Design decisions, e.g., language, platform, API, etc., chosen for prototype are difficult to have changed, but may be inappropriate for completed system

Small, visible changes between prototype and finished system are easily perceived by the customer

– Rapid Application Development

High-speed modification of linear sequential mode.

Component-based construction of system

Very short time frame

Typically used for information systems

Difficult for applications in which the parts are not already components

Unsuited for projects with high technical risk

Evolutionary Process Models

All software evolves (changes) over time

Requirements change over the lifetime of the project

Time to market means we cannot wait until the very end of the project for a solution

Must make efficient use of team members

Iterative model

Develop increasingly more complex versions of the software

– Incremental Model

Combines linear sequential model with prototyping

Produces increments of a system.

First produce the core product

A set of new functionality is added in each new increment

The first increment can be viewed as a prototype that is used by the client

Overlapping sequences of process stages

Focus on a set of deliverables

Allows workers dedicated to a particular stage, e.g., “short, wide” developers

– Spiral Model

Software is developed in a set of incremental releases

Early iterations may be prototypes or paper models

Later iterations are increasingly more complex versions of the software

Divided into a number of framework activities or task regions (typically between 3 and 6)

Allows for efficient use of resources

– Component Assembly Model

Use a set of pre-existing components to construct a new system

Need a library of existing component

Need a method of indexing these components

Narrow domain

Subset of system uses existing components

– Concurrent Development Model

• Phases of software lifecycle

Definition Phase – determine behavior of the system

Development Phase – determine how to obtain the desired behavior

Maintenance - change the behavior

– Corrective - fix uncovered defect

– Adaptive - Platform change

– Enhancement - Perfective, additional functionality

– Preventive - re-engineering, make system more maintainable

• UML – use case diagrams, class diagrams, sequence diagrams

Use Case Diagram – high-level behaviors of the system, user goals, external entities: actors

Describes a set of sequences.

Each sequence represents the interactions of things outside the system (actors) with the system itself (and key abstractions)

Use cases represent the functional requirements of the system (non-functional requirements must be given elsewhere)

Each use case has a descriptive name

Describes what a system does but not how it does it.

Use case names must be unique within a given package

Examples: withdraw money, process loan

Class Diagram – set of classes and their relationships. Describes interface to the class (set of operations describing services)

Sequence Diagram – focus on time ordering of messages

X-axis is objects – Object that initiates interaction is left most – Object to the right are increasingly more subordinate

Y-axis is time – Messages sent and received are ordered by time

Object life lines represent the existence over a period of time • Activation (double line) is the execution of the procedure.

• Differences between boundary, control, and entity classes

Boundary – represent the interactions between the system and actors

Control – represent the tasks that are performed by the user and supported by the system

Entity – represent the persistent information tracked by the system

• Refactoring (definitions and the basic idea)

fac·tor – The individual items that combined together form a complete software system:

identifiers

contents of function

contents of classes and place in inheritance hierarchy

fac·tor·ing – Determining the items, at design time, that make up a software system

Process of changing a software system in such a way that it does not alter the external behavior of the code, yet improves its internal structure

A program restructuring operation to support the design, evolution, and reuse of object oriented frameworks that preserve the behavioral aspects of the program

High Level - – Features to be added to a system – e.g., New a new menu, or menu item

Intermediate Level – Change design (factoring) – e.g., Move a member function

Low Level – Change lines of code – e.g., Changes in (a least) two classes

Some Code Smells: A symptom in code of a possible deeper problem

Not bugs or errors

Not technically incorrect

Problems that are good candidates to be refactored to improve comprehensibility and longer term maintainability

Duplicated code: identical or very similar code exists in more than one location.

Long method: a method, function, or procedure that has grown too large.

Large class: a class that has grown too large, aka god class

Too many parameters: a long list of parameters in a procedure or function make readability and code quality worse

Feature Envy: a class that uses methods of another class excessively.

Inappropriate Intimacy: a class that has dependencies on implementation details of another class.

Refused Bequest: a class that overrides a method of a base class in such a way that the contract of the base class is not honored by the derived class.

• Extreme programming (definitions and the basic idea) -- Agile

Extreme Programming (XP) is a (very) lightweight incremental software development process.

It involves a high-degree of discipline from the development team

Comprised of 12 core practices

Planning

Small Releases

System Metaphor

Simple Design

Continuous Testing

Refactoring

Pair Programming

Collective Code Ownership

Continuous Integration

40-Hour Work Week

On-site Customer

Coding Standards

Most novel aspect of XP (as a process) is the use of pair programming

Rapid feedback – from customer

Assume simplicity – keep designs simple

Incremental change – small changes keep things manageable

Embracing change – keep your options open

Quality work – strive for high quality products

• Estimation (basic ideas)

Estimating models in other fields

large base of history

in wide use

generate detailed planning data

require a size estimate as input

Software size estimating experience

100% + errors are normal

few developers make estimates

fewer still use orderly methods

Estimating is an uncertain process.

no one knows how big the product will be

the earlier the estimate, the less is known

estimates can be biased by business and other pressures

Estimating is an intuitive learning process.

ability improves with experience

some people will be better at estimating than others

Estimating is a skill.

improvement will be gradual

you may never get very good

The objective, however, is to get consistent.

you will then understand the variability of your estimates

you seek an even balance between under and over estimates

The principal advantages of using a defined estimating method are

you have known practices that you can work to improve

it provides a framework for gathering estimating data

by using consistent methods and historical data, your estimates will get more consistent

Standard Component Method

Gather data about various level of program abstraction, subsystems, modules, reports, screens.

Compare these to what is predicted in the system

Estimate= Smallest value estimate + 4\*(Most likely or common estimate) + Largest value estimate

Function Point

Functions:

Inputs: screens, forms (UI) or other programs which add data to the system. Inputs that require unique processing

Outputs: Screens, reports, etc

Inquiries: Screens which allow users to interrogate or ask for assistance or information

Data files: logical collections of records, tables in a DB

Interfaces: Shared files, DB, parameters lists

Review requirements:

Count number of each function point type

Use historical data on each function point type to determine estimate

Function point does not map to physical part of source.

Can not measure FP in a given system (automatically)

Proxy Based Estimation

Based on previous projects that are similar to the prospective project

• Coupling, cohesion, software metrics

Cohesion – Internal interaction of the module. Crisp abstraction of purpose

In order from good (high) to bad (low)

– Informational

Performs a number of actions

Each action has its own entry point and independent code

All actions are performed on a shared data structure

Object-Oriented

– Functional

Module that performs a single action or achieves a single goal

Maintenance involves the entire single module

Very reusable because the module is completely independent in action of other modules

Can be replaced easily

– Communicational

Action based on the ordering of steps on all the same data

Actions are related but still not completely separated – Module update record in database and write it to the audit trail – Module calculate new trajectory and send it to the printer

Module cannot be reused

– Procedural

Action based on the ordering of steps

Related by usage in ordering – Module read part number from an input file and update directory count

Changes to the ordering of steps or purpose of steps requires changing the module abstraction

Limited situations where this particular sequence is used is limited

– Temporal

Modules performs a series of actions that are related by time

Often happens in initialization or shutdown

Degrades to temporal cohesion if time of action changes

Addition of parts to the system may require additions to multiple modules

– Logical

Module performs a series of related actions, one of which is selected by the calling module

Parts of the module are related in a logical way, but not the primary logical association

May include high and low-level actions in the same module

May include unused parameters for certain uses

Difficult to understand interface (in order to do something you have to wade through a lot of unrelated possible actions)

– Coincidental

Performs multiple, completely unrelated actions

May be based on factors outside of the design, i.e., skillset or interest of developers, avoidance of small modules

No reusability

Poor correct maintenance and enhancement

Break into smaller modules

Coupling – External interaction of the module with other modules

In order from good (low) to bad (high)

– Data Coupling

Only required data is passed from one module to another

All arguments are homogenous data items

– simple data type

– complex data type, but all parts are used

Holy grail

Allows for reuse, maintenance, understanding, etc

– Stamp Coupling

One module passes more data then needed to another module

– void swap(int v[], int i, int j);

– double calcsalary(Employee& e);

Often involves records (structs) with lots of fields

Entire record is passed, but only a few fields are used

Efficiency considerations?

– Control Coupling

One module passes an element of control to another module

One module explicitly controls the logic of another

– Control switch is passed as an argument

– Module p passes an argument to module q that directly tells it what control structure path to take

Control coupling?

– Module p calls module q and q passes a flag back to p that indicates an error

– Module p calls module q and q passes a flag back to p that tells p that it must output the error “I goofed up” \*\*\*\*\* control coupling

Modules should pass data and leave control path decisions private to a module

Independent reuse is not possible

– Common Coupling

Using global variables

All modules have read/write access to a global data block

Modules exchange data using the global data block (instead of arguments)

Single module with write access where all other modules have read access is not common coupling

Have to look at many modules to determine the current state of a variable

– Side effects require looking at all the code in a function to see if there are any global effects

– Changes in one module to the declaration requires changes in all other modules

– Identical list of global variables must be declared for module to be reused

– Module is exposed to more data than is needed

– Content Coupling

A module directly references the content of another module

– Module p modifies a statement of module q

– Module p refers to local data of module q (in terms of a numerical displacement)

– Module p branches to a local label of module q

Content coupled modules are inextricably interlinked

– Change to module p requires a change to module q (including recompilation)

– Reusing module p requires using module q also

Software Metrics

Measure - quantitative indication of extent, amount, dimension, capacity, or size of some attribute of a product or process.

e.g. Defect Rates, Error Rates

Metric - quantitative measure of degree to which a system, component or process possesses a given attribute.

e.g. Errors per person per hour -> Gives a good frame of reference for the Measures (above)

Number of errors

Number of errors found per person hours expended

Metric: A handle or guess about a give attribute.