INTRODUCTION TO SOFTWARE ENGINEERING

LECTURE – 25

MAY 22, 2017

TOPICS COVERED

- Distributed Software Engineering
- Reuse Based Software Engineering
- Configuration Management

- **Testing Conventional Applications**

QUICK LOOK

- **Once source code has been generated, software must be tested to uncover as many errors as possible**
- **During early stages of testing, a software engineer performs all tests**
- **Reviews and other SQA (Software Quality Assurance) activities can do and uncover errors, but they are not sufficient**
- **Software is tested from two different perspectives:**
  - Internal program logic is exercised using white-box test cases design techniques
  - Software requirements are exercised using black-box test-case design techniques

QUICK LOOK

- **The work product**
  - A set of test cases designed to exercise internal logic, interfaces, component collaborations, and external requirements is designed and documented
  - expected results are defined and actual results are recorded
- **Try hard to break the software**
- **Design test cases in a disciplined fashion and review then for thoroughness**

TESTING FUNDAMENTALS

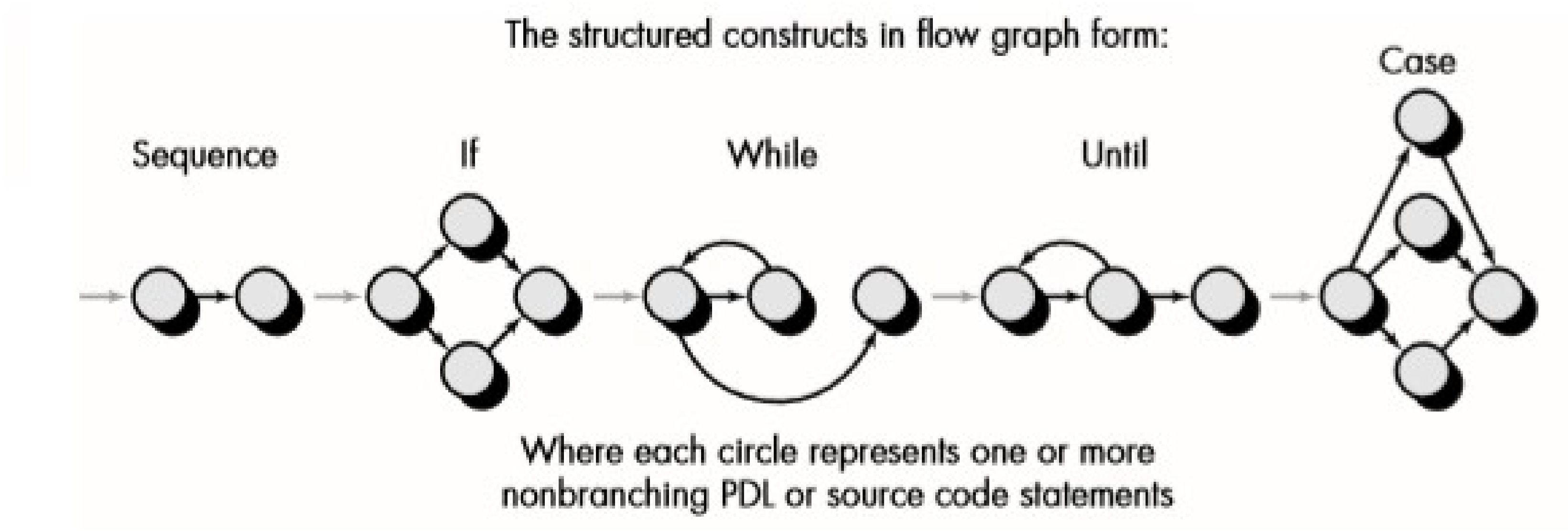
- **The goal of testing is to find errors and a good test is one that has a high probability of finding an error**
- **Therefore, you should design and implement a system or a product with “testability” in mind**
- **Testability**
  - It is simply how easily a computer program can be tested
- **Operability**
  - The better it works, the more efficiently it can be tested
- **Observability**
  - What you see is what you test

TESTING FUNDAMENTALS

- **Controllability**
  - The better we can control the software, the more the testing can be automated and optimized
  - All possible outputs can be generated through some combination of input
  - Software and hardware states and variables can be controlled directly by the test engineer

PATH TESTING

- **Flow Graph Notation**



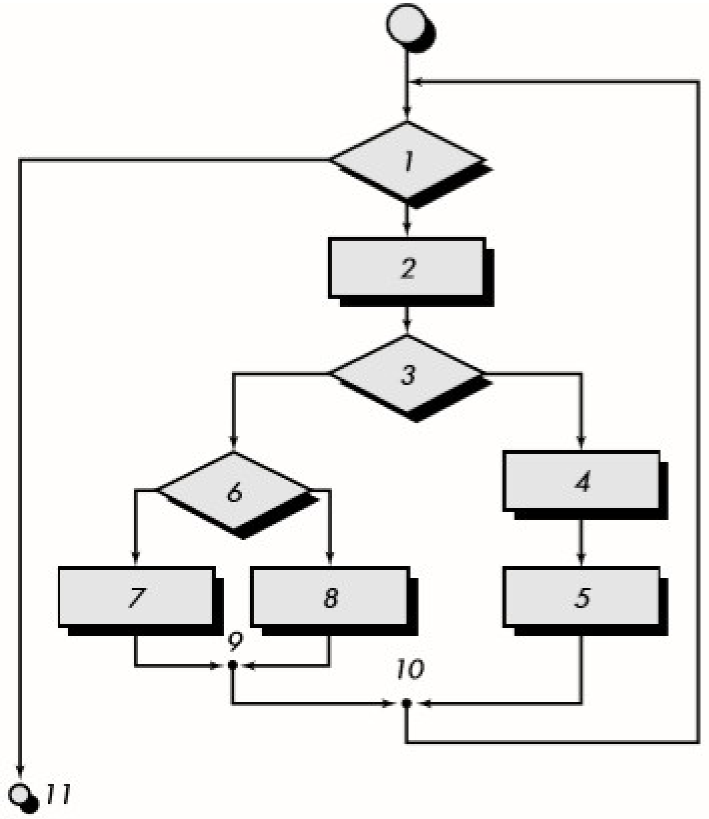
The structured constructs in flow graph form:

Where each circle represents one or more nonbranching PDL or source code statements

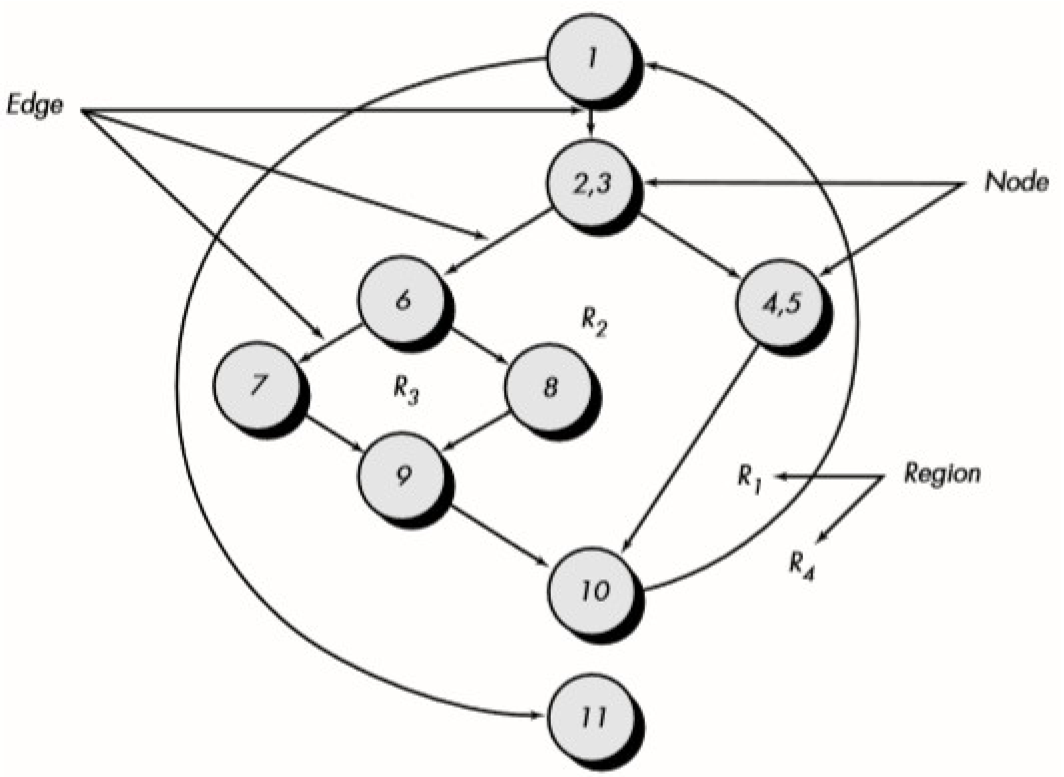
CYCLOMATIC COMPLEXITY

- **Cyclomatic complexity is a software metric that provides a quantitative measure of the logical complexity of a program**
- **The value computed for cyclomatic complexity defines the number of independent paths in the basis set of a program**
- **It provides us with an upper bound for the number of tests that must be conducted to ensure that all statements have been executed at least once**
- **An independent path is any path through the program that introduces at least one new set of processing statements or a new condition**

FLOW CHART AND FLOW GRAPH



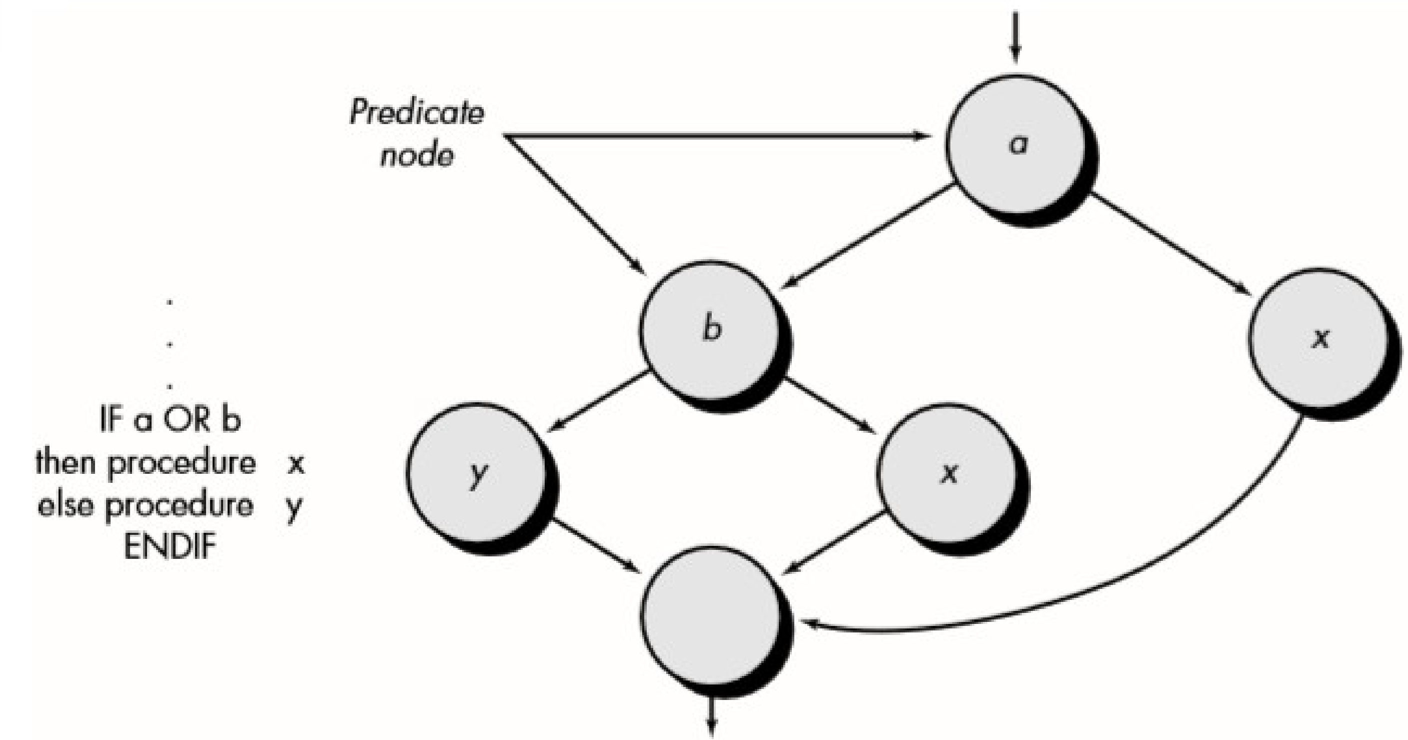
(A)



(B)

FLOW CHART AND FLOW GRAPH

- **Compound Logic**



```
.
.
.
IF a OR b
then procedure   x
else procedure   y
ENDIF
```

INDEPENDENT PROGRAM PATHS

- **It is any path through the program that introduces at least one new set of processing statements or a new condition**
- **path 1: 1-11**
- **path 2: 1-2-3-4-5-10-1-11**
- **path 3: 1-2-3-6-8-9-10-1-11**
- **path 4: 1-2-3-6-7-9-10-1-11**
- **The number of regions of the flow graph correspond to the cyclomatic complexity**
- **Cyclomatic complexity, V(G), for a flow graph, G, is defined as V(G) = E - N + 2**
- **where E is the number of flow graph edges, N is the number of flow graph nodes**

INDEPENDENT PROGRAM PATHS

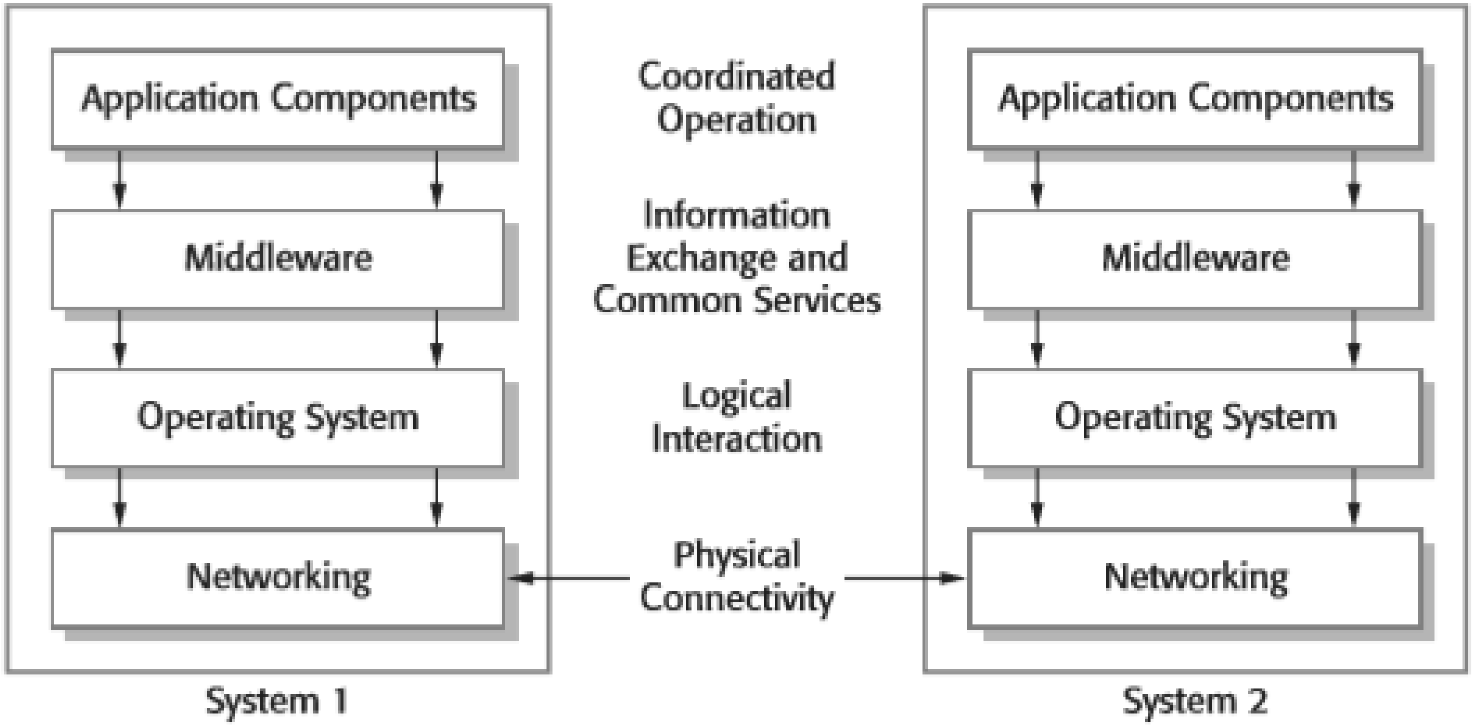
- **Cyclomatic complexity, V(G), for a flow graph, G, is also defined as**
  - V(G) = P + 1
  - where P is the number of predicate nodes contained in the flow graph G
- **The cyclomatic complexity can be computed using each of the algorithms just noted:**
  - The flow graph has four regions
  - V(G) = 11 edges - 9 nodes + 2 = 4
  - V(G) = 3 predicate nodes + 1 = 4
- **The value for V(G) provides us with an upper bound for the number of independent paths that form the basis set**
  - an upper bound on the number of tests that must be designed and executed to guarantee coverage of all program statements

- **Distributed Software Engineering**

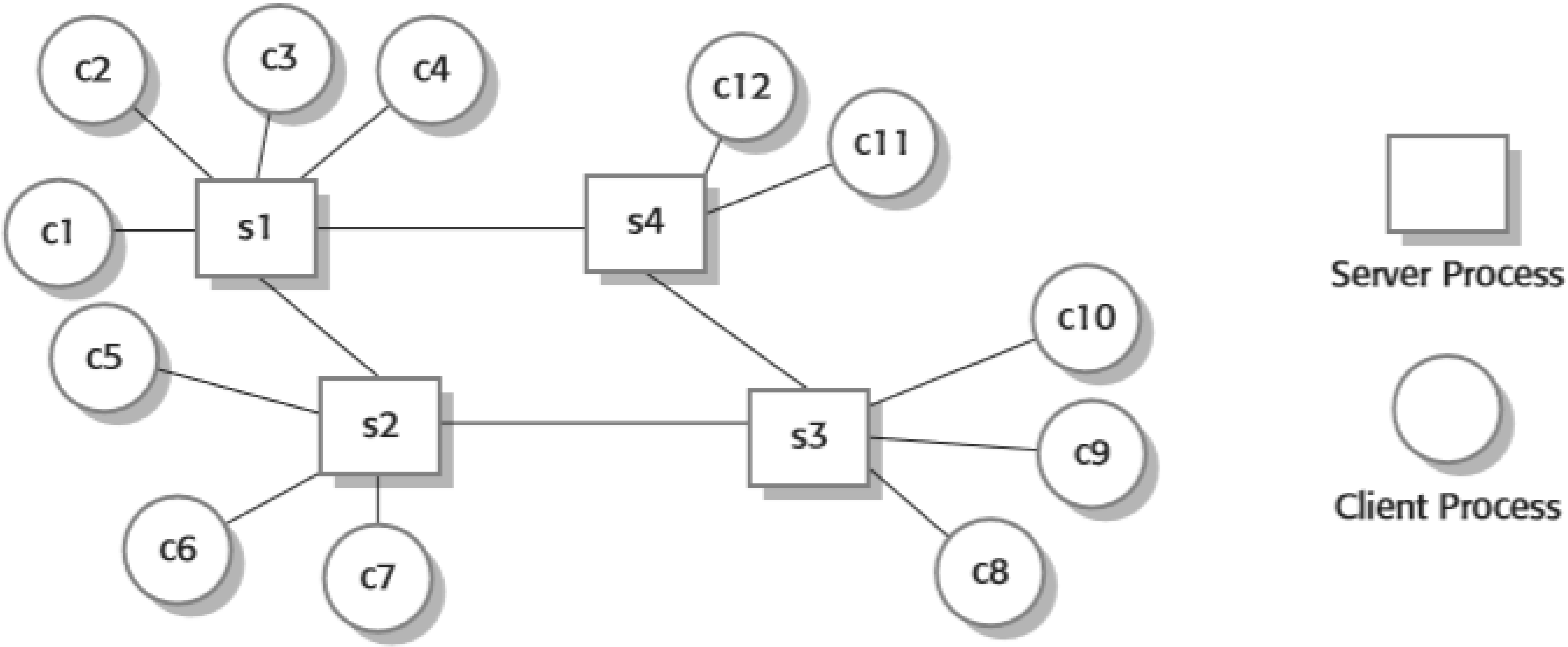
DISTRIBUTED SYSTEMS ISSUES

- **Transparency**
- **Openness**
- **Scalability**
- **Security**
- **Quality of Service**
- **Failure Management**

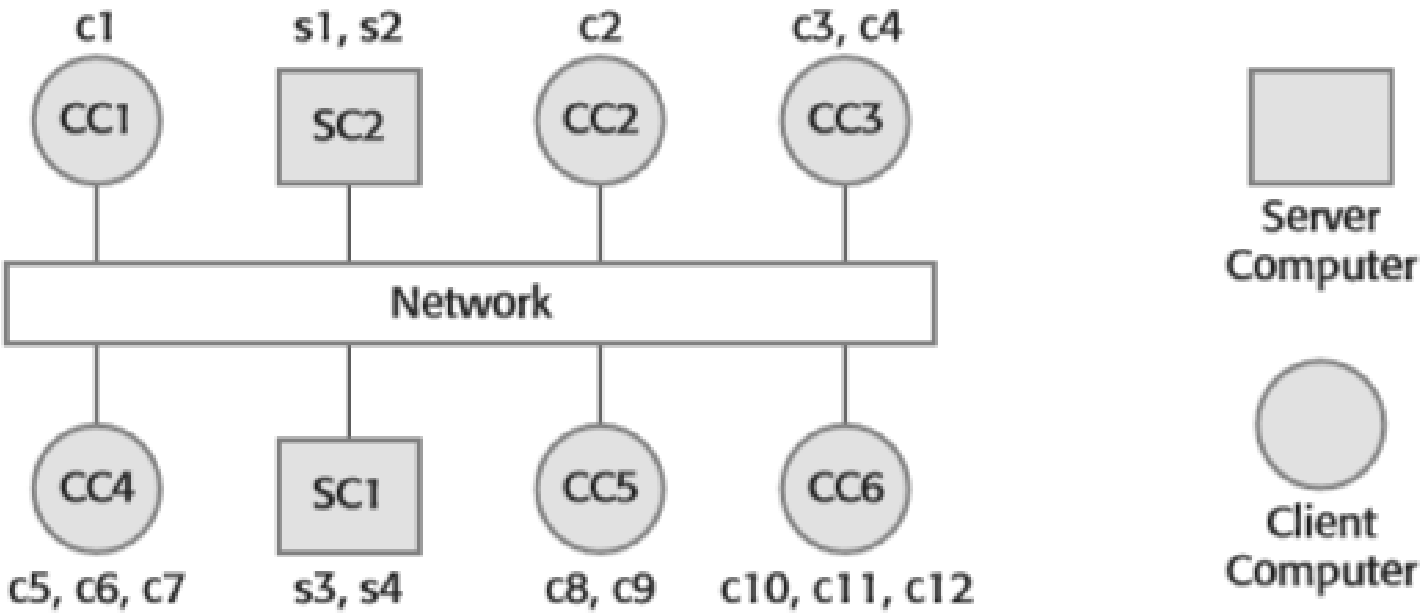
MIDDLEWARE


Application Components
Middleware
Operating System
Networking
System 1
Coordinated Operation
Information Exchange and Common Services
Logical Interaction
Physical Connectivity
Application Components
Middleware
Operating System
Networking
System 2

CLIENT-SERVER INTERACTION



CLIENT-SERVER COMPUTING



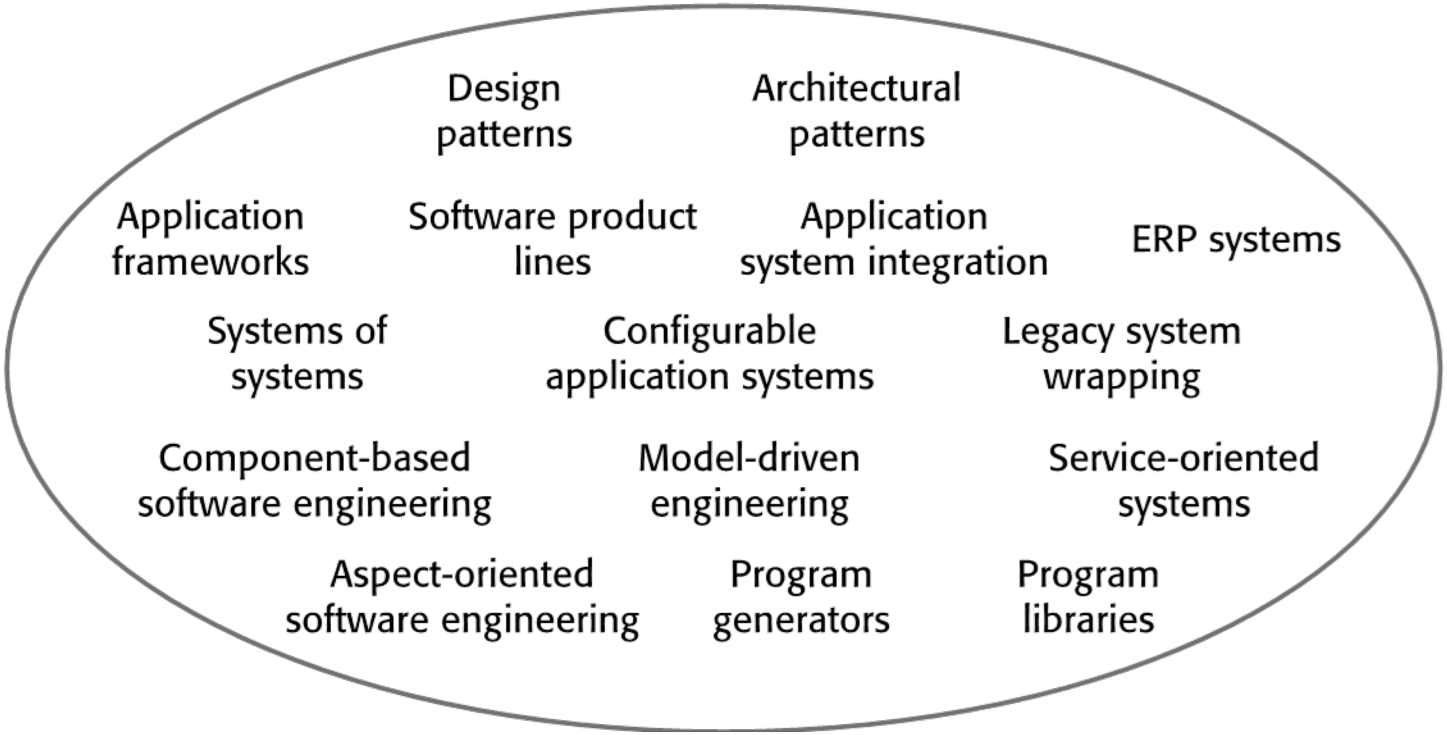
REUSE-BASED SOFTWARE ENGINEERING

- **System reuse**
  - Complete systems, which may include several application programs may be reused.
- **Application reuse**
  - An application may be reused either by incorporating it without change into other or by developing application families.
- **Component reuse**
  - Components of an application from sub-systems to single objects may be reused.
- **Object and function reuse**
  - Small-scale software components that implement a single well-defined object or function may be reused.

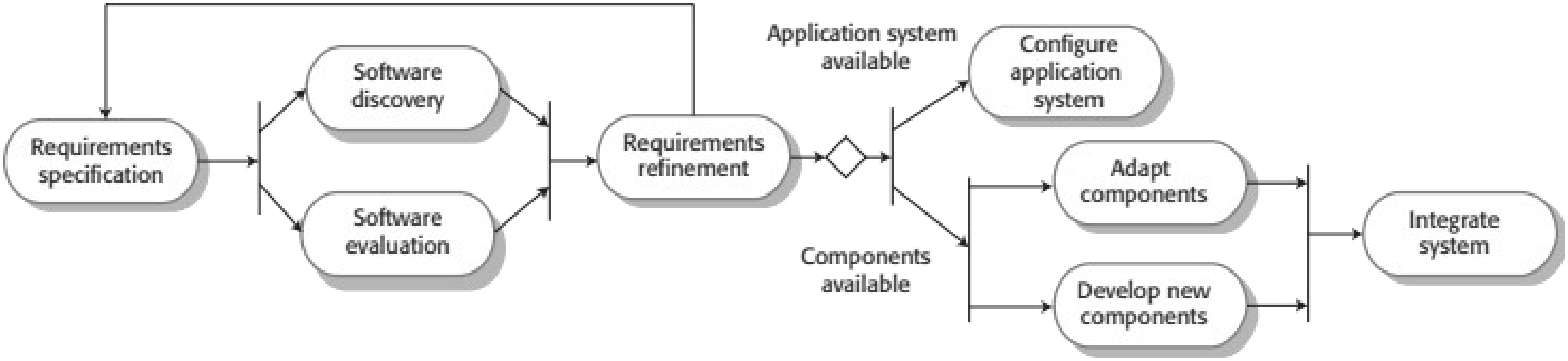
THE RE-USE LANDSCAPE

- **Although reuse is often simply thought of as the reuse of system components, there are many different approaches to reuse that may be used**
- **Reuse is possible at a range of levels from simple functions to complete application systems**
- **The reuse landscape covers the range of possible reuse techniques**

THE RE-USE LANDSCAPE



REUSE-ORIENTED SOFTWARE ENGINEERING



KEY PROCESS STAGES FOR ACQUISITION

- **Requirements specification**
- **Software discovery and evaluation**
- **Requirements refinement**
- **Application system configuration**
- **Component adaptation and integration**

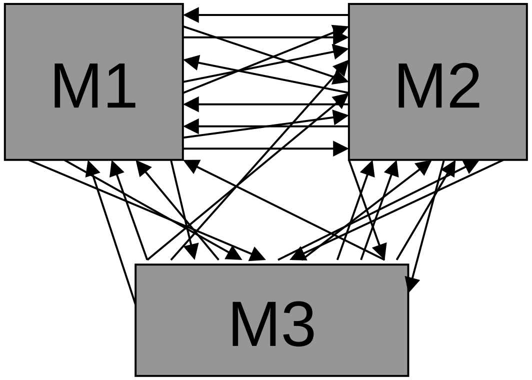
DOMAIN ENGINEERING FOR REUSE IN SE

Domain Engineering entails:

- **Domain Analysis**
  - Commonalities and differences of systems in a domain are discovered and recorded
- **Domain Implementation**
  - It means the use of information collected in domain analysis to create reusable components and new systems

CHARACTERISTICS OF GOOD DESIGN

- **Component independence**
  - High cohesion
  - Low coupling
- **Exception identification and handling**
- **Fault prevention and fault tolerance**
- **Design for change**

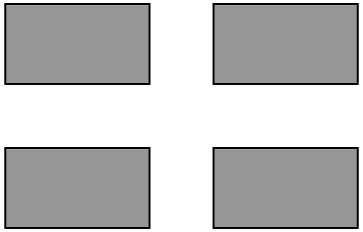


COHESION

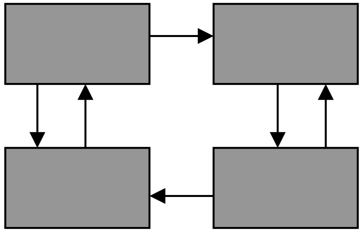
- **Definition**
  - The degree to which all elements of a component are directed towards a single task
  - The degree to which all elements directed towards a task are contained in a single component
  - The degree to which all responsibilities of a single class are related
- **Internal glue with which component is constructed**
- **All elements of component are directed toward and essential for performing the same task**

COUPLING

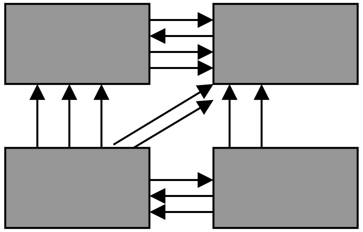
- **The degree of dependence such as the amount of interactions among components**



No dependencies



Loosely coupled
some dependencies



Highly coupled
many dependencies

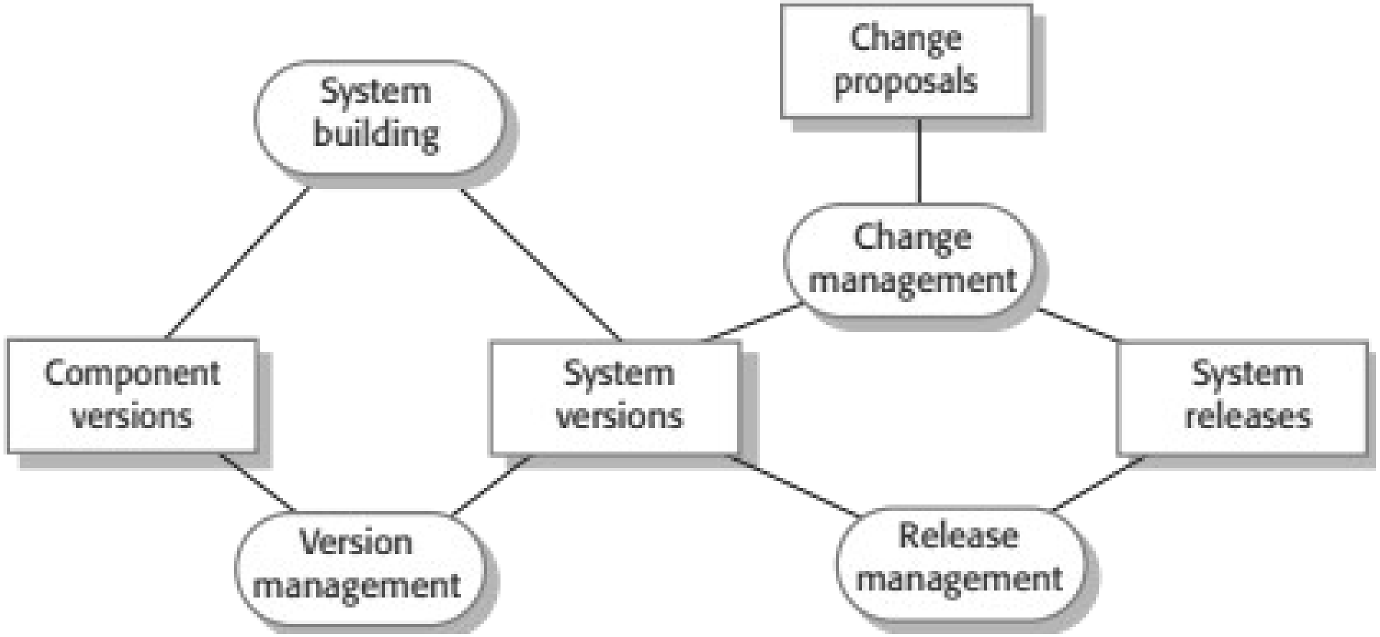
CONFIGURATION MANAGEMENT (CM)

- **Software systems are constantly changing during development and use**
- **Configuration management (CM) is concerned with the policies, processes and tools for managing changing software systems**
- **You need CM because it is easy to lose track of what changes and component versions have been incorporated into each system version**
- **CM is essential for team projects to control changes made by different developers**

CM ACTIVITIES

- **Version management**
  - Keeping track of the multiple versions of system components and ensuring that changes made to components by different developers do not interfere with each other
- **System building**
  - The process of assembling program components, data and libraries, then compiling these to create an executable system
- **Change management**
  - Keeping track of requests for changes to the software from customers and developers, working out the costs and impact of changes, and deciding the changes should be implemented
- **Release management**
  - Preparing software for external release and keeping track of the system versions that have been released for customer use

CM ACTIVITIES



AGILE DEVELOPMENT AND CM

- **Agile development, where components and systems are changed several times per day, is impossible without using CM tools**
- **The definitive versions of components are held in a shared project repository and developers copy these into their own workspace**
- **They make changes to the code 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

Q&A