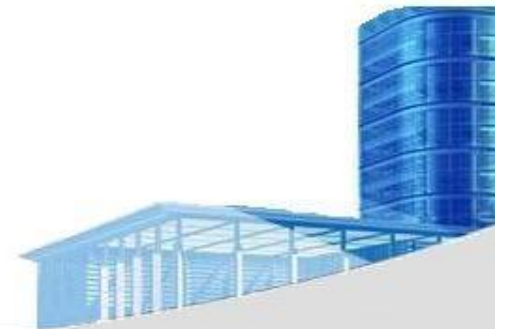




# Ch.29 **S**oftware **C**onfiguration **M**anagement (**C**ont.)

June 1,2015





## • The **SCM** Process

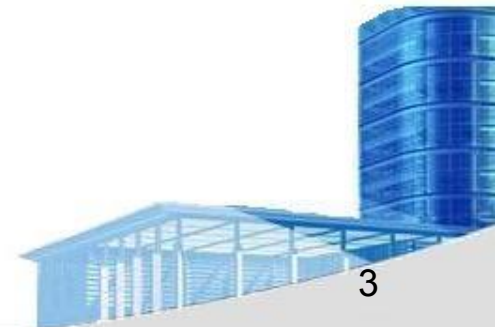
1. **Identification**: How does an organization identify and manage the many existing versions of a program (and its documentation) in a manner that will **enable change** to be accommodated efficiently?
2. **Version Control**: How does an organization control changes before and after software is released to a customer?
3. **Change Control**: Who has responsibility for approving and ranking changes?
4. **Configuration Auditing**: How can we ensure that changes have been made properly?
5. **Reporting**: What mechanism is used to appraise others of changes that are made?





- **SCM Repository**(中心存储库)

- The SCM repository is the set of **mechanisms and data structures** that allow a software team to **manage change** in an **effective manner**
- The repository performs or **precipitates** (引发) the following functions [**For89**]:
  - Data integrity
  - Information sharing
  - Tool integration
  - Data integration
  - Methodology **enforcement**(实施)
  - Document standardization





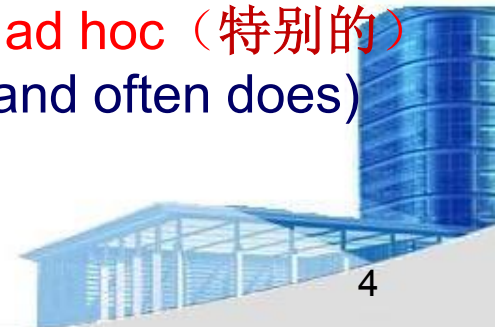
- **SCM for Web and Mobile Engineering - I**

- *Content.*

- A typical Web or Mobile App contains a vast array of content—text, graphics, **applets** (**Java程序**), scripts, audio/video files, forms, active page elements, tables, streaming data, and many others.
- The **challenge** is to organize this sea of content into a **rational set of configuration objects** (Section 29.2.1) and then establish appropriate configuration control mechanisms for these objects.

- *People.*

- Because a significant percentage of Web and Mobile App development continues to be conducted in an **ad hoc** (**特别的**) manner, **any person involved** in the App can (and often does) **create content**.





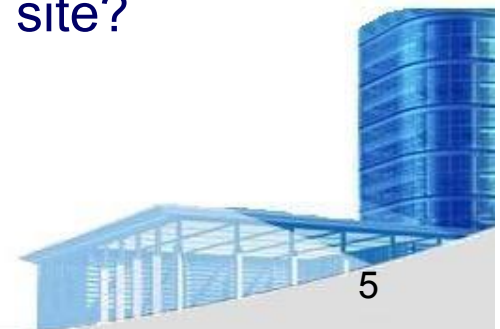
- **SCM for Web and Mobile Engineering - II**

- *Scalability.*

- As **size and complexity grow**, small changes can have **far-reaching and unintended affects** that can be problematic. Therefore, the rigor of configuration control mechanisms should be directly proportional to application scale.

- *Politics.*

- Who **'owns'** a App?
- Who assumes **responsibility** for the **accuracy** of the information displayed by the App?
- Who assures that **quality control** processes have been followed before information is published to the site?
- Who is responsible for **making changes**?
- Who assumes the **cost of change**?





# • Content Management - I

- **The collection subsystem** encompasses all actions required to create and/or acquire content, and the technical functions that are necessary to
  - convert content into a form that can be represented by a mark-up language (e.g., HTML, XML)
  - organize content into packets that can be displayed effectively on the client-side.
- **The management subsystem** implements a repository that encompasses the following elements:
  - **Content database**—the information structure that has been established to store all content objects
  - **Database capabilities**—functions that enable the CMS (Content Management System) to search for specific content objects (or categories of objects), store and retrieve (恢复) objects, and manage the file structure that has been established for the content
  - **Configuration management functions**—the functional elements and associated workflow that support content object identification, version control, change management, change auditing, and reporting.





## • Content Management - II

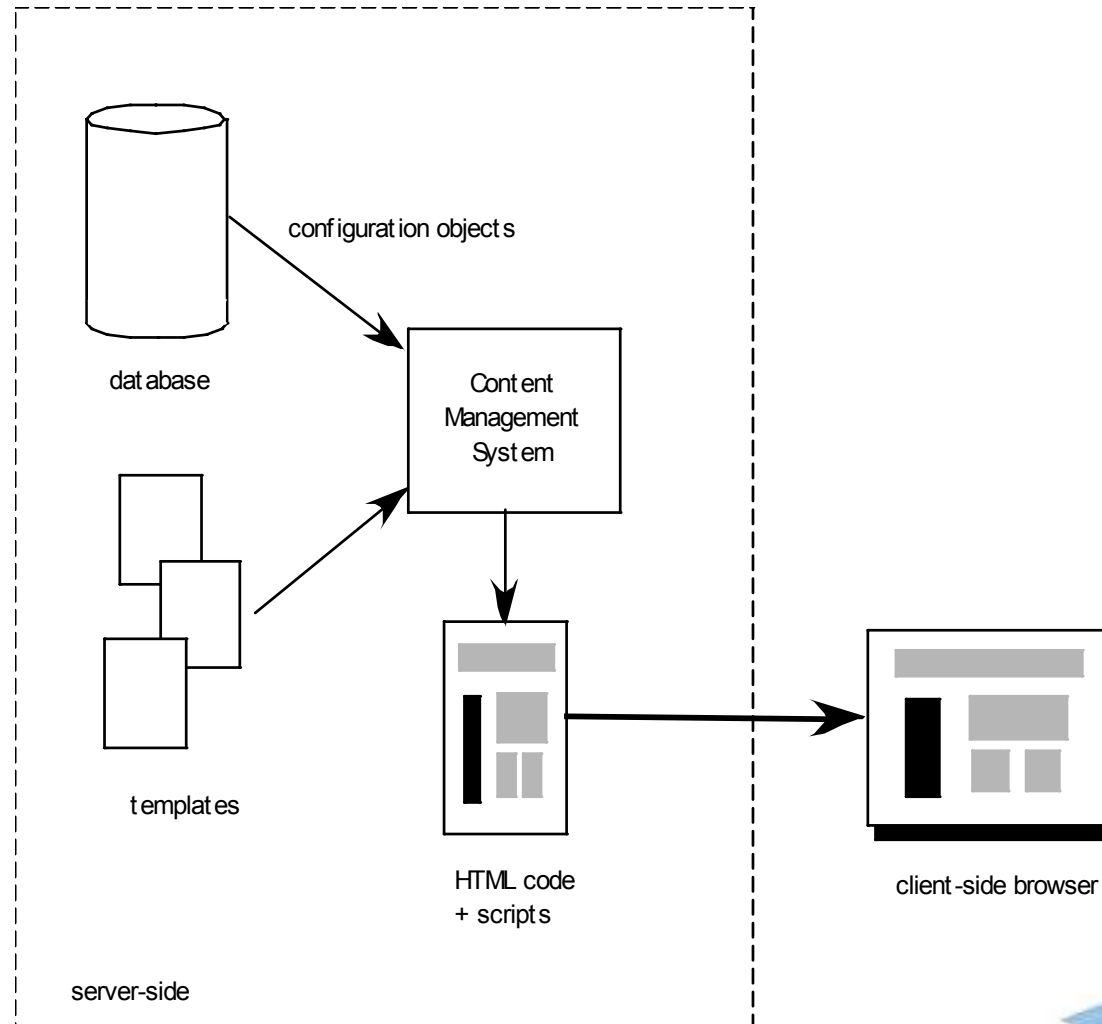
- **The *publishing subsystem*** extracts from the repository, converts it to a form that is amenable to publication, and formats it so that it can be transmitted to client-side browsers. The publishing subsystem accomplishes these tasks using a series of templates.
- Each **template** is a function that builds a publication using one of **three different components** [BOI02]:
  - **Static elements**—text, graphics, media, and scripts that require no further processing are transmitted directly to the client-side
  - **Publication services**—function calls to specific **retrieval** (检索) and formatting services that personalize content (using predefined rules), perform data conversion, and build appropriate navigation links.
  - **External services**—provide access to external corporate information infrastructure such as enterprise data or “back-room” applications.







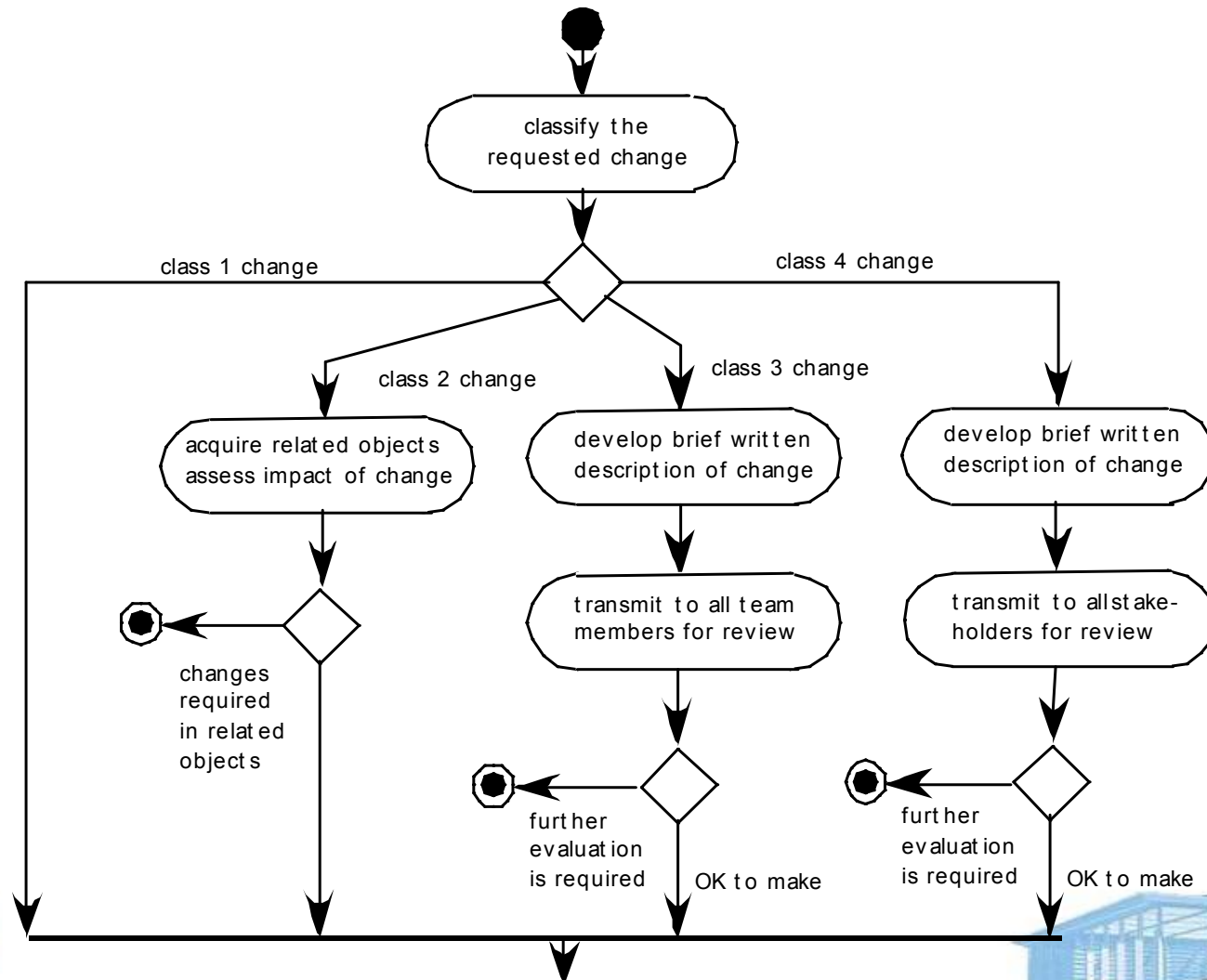
- Content Management - III





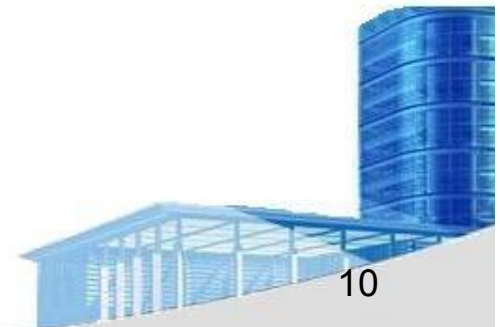
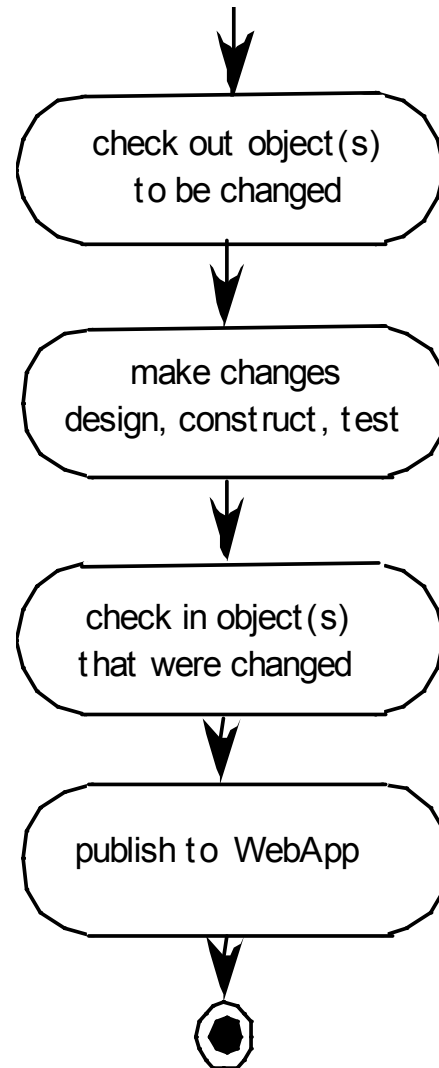


- Change Management for Web and Mobile Apps - I





- **Change Management for Web and Mobile Apps - II**





# Ch.23 Testing Conventional Applications





- **Testability**

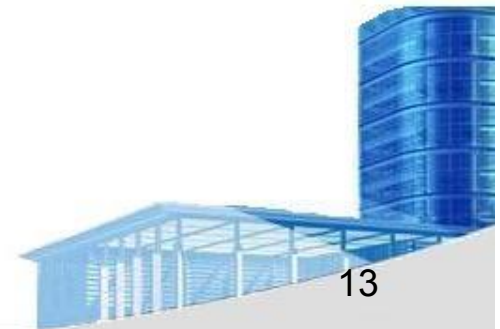
- **Operability** - the better it works the more efficiently it can be tested
- **Observability** - what you see is what you test
- **Controllability** - the better software can be controlled the more testing can be automated and optimized
- **Decomposability** - by controlling the scope of testing, the more quickly problems can be isolated and retested intelligently
- **Simplicity** - the less there is to test, the more quickly we can test
- **Stability** - the fewer the changes, the fewer the disruptions to testing
- **Understandability** - the more information known, the smarter the testing





- **What is a “Good” Test?**

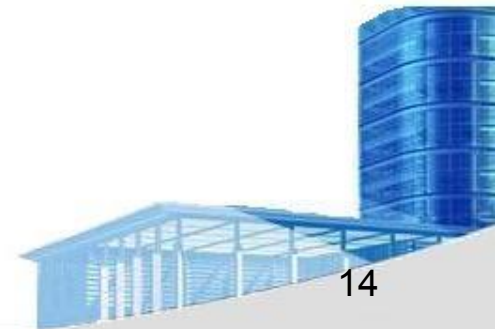
- A good test has a **high probability** of finding an error
- A good test is **not redundant**.
- A good test should be “best of **breed**”
- A good test should be **neither too simple nor too complex**





- **Internal and External Views**

- Any engineered product (and most other things) can be tested in one of two ways:
  - Knowing the specified function that a product has been designed to perform, tests can be conducted that demonstrate each function is fully operational while at the same time searching for errors in each function;
  - Knowing the internal workings of a product, tests can be conducted to ensure that "all gears mesh," that is, internal operations are performed according to specifications and all internal components have been adequately exercised.

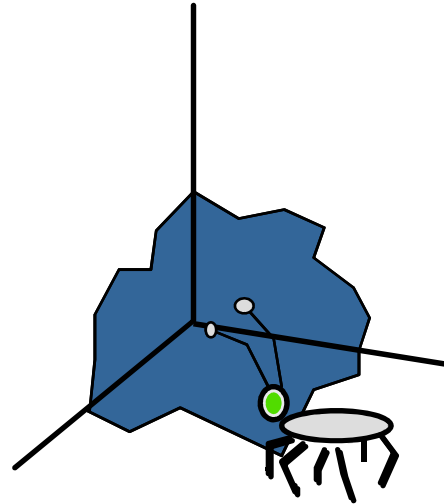




- Test Case Design

"Bugs lurk in corners  
and congregate at  
boundaries ..."

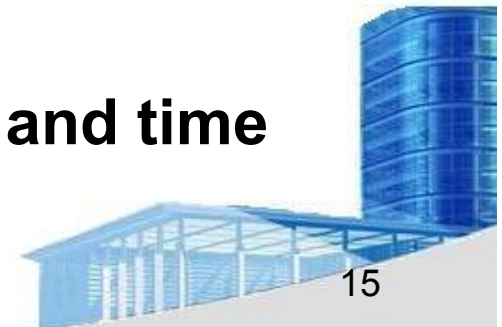
*Boris Beizer*



**OBJECTIVE** to uncover errors

**CRITERIA** in a complete manner

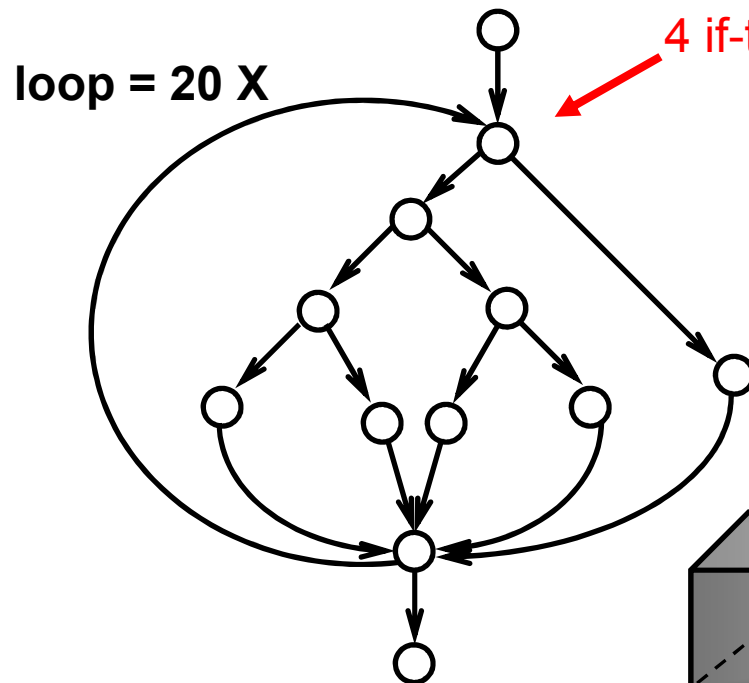
**CONSTRAINT** with a minimum of effort and time







- Why **not** try **exhaustive** testing?



There are  $5^{20} \approx 10^{14}$  possible paths! If we execute one test per millisecond, it would take **3,170 years** to test this program!!

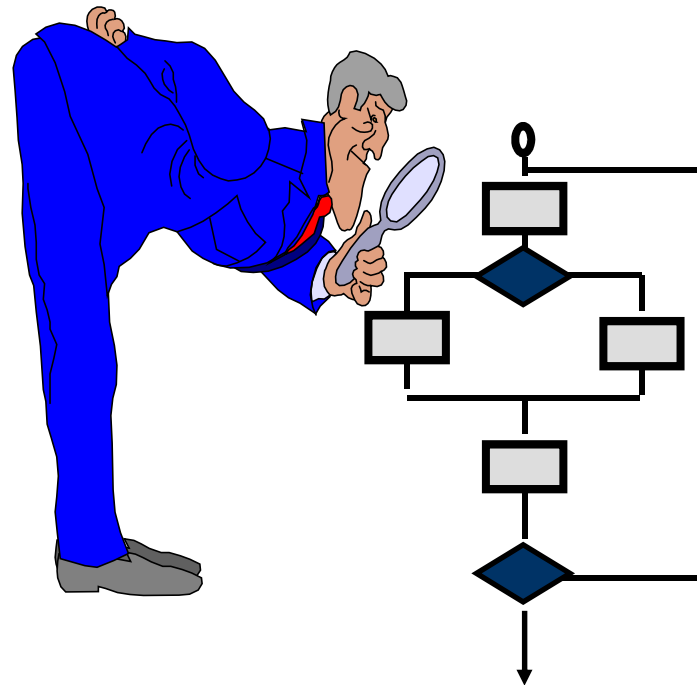
White-Box Testing

Black-Box Testing

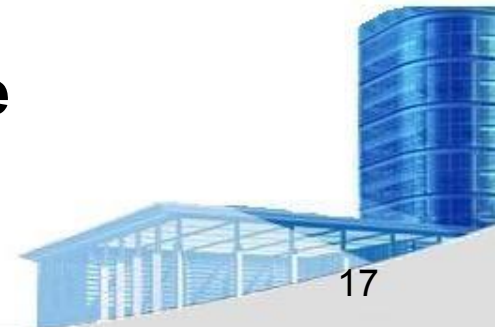
Note: Testing **cannot** show the **absence** of errors and defects.



- **White-Box Testing**



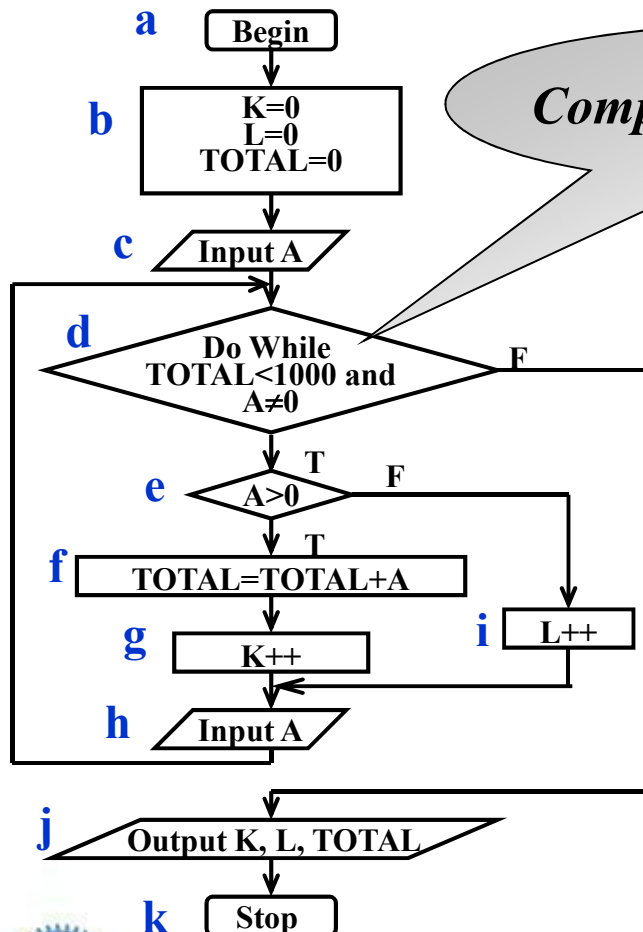
... our goal is to ensure that **all statements and conditions** have been executed at least once ...



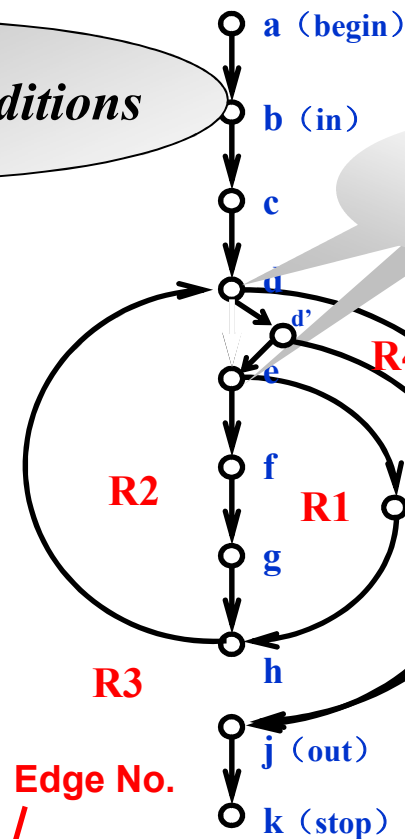


# Basis Path Testing

- **Goal:** execute every statement at least once



*Compound conditions*



**Predicate node**

E=14

N=12

V(G)=4

P=3

Edge No.

# of Regions (域数)

$E - N + 2$

$P + 1$

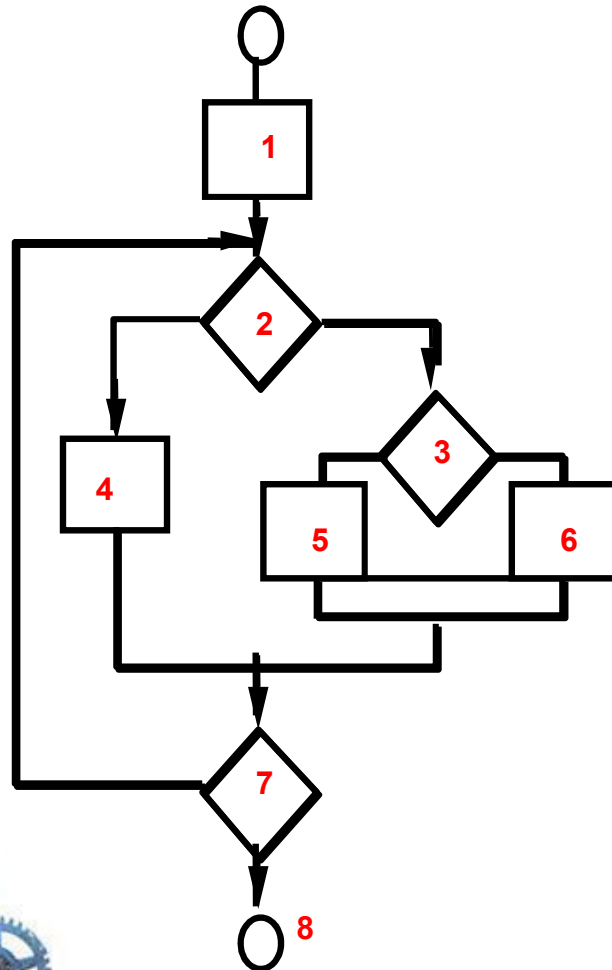
Node No.

Predicate Node No.

Cyclomatic Complexity V(G)=  
(环复杂度)



- Basis Path Testing**



Next, we derive the independent paths:

Since  $V(G) = 4$ , there are four paths

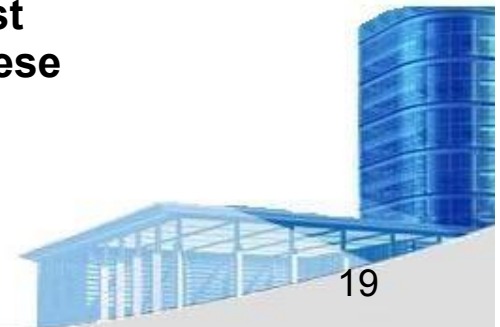
Path 1: 1,2,3,6,7,8

Path 2: 1,2,3,5,7,8

Path 3: 1,2,4,7,8

Path 4: 1,2,4,7,2,4,...7,8

Finally, we derive test cases to exercise these paths.

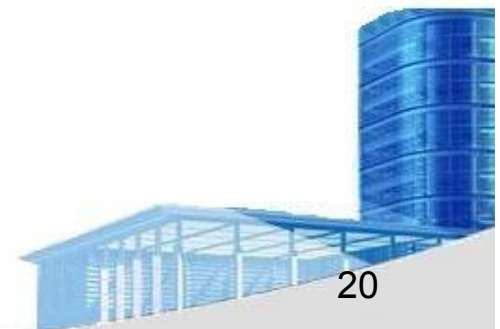




- **Deriving Test Cases**

- Summarizing:

- Using the design or code as a foundation, draw a corresponding flow graph.
- Determine the cyclomatic complexity ( $V(G)$ ) of the resultant flow graph.
- Determine a basis set of linearly independent paths.
- Prepare test cases that will force execution of each path in the basis set.





## Control Structure Testing

- **Data Flow Testing:** selects test paths according to the locations of variable **definitions** and **uses** in the program

**DEF(S)** = { X | **S** contains a definition of X }; **X=1; B=2**

**USE(S)** = { X | S contains a use of X }. **X=X+B ; C=X**

**DU chain of X** = [X, S, S'] where  $X \in \text{DEF}(S)$  and  $X \in \text{USE}(S')$ , and the definition of X in S is live at S' ---- the flow of X.

Every DU chain is covered once.

e.g. X = [X, X=1, C=X ]

**S**      **S'**

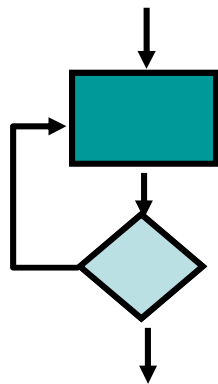
**If X>1 Then ..... Else .....**





# Control Structure Testing

- Loop Testing



**Simple loop**

0

1

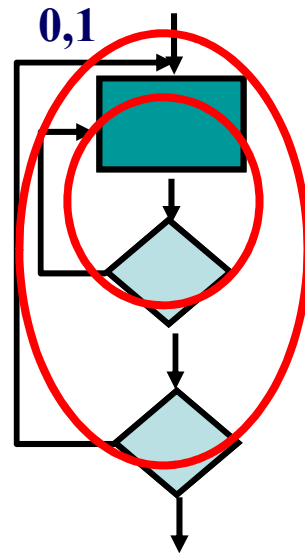
2

$2 < m < n$

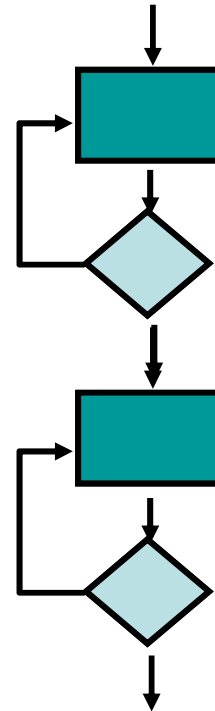
$n-1$

$n$

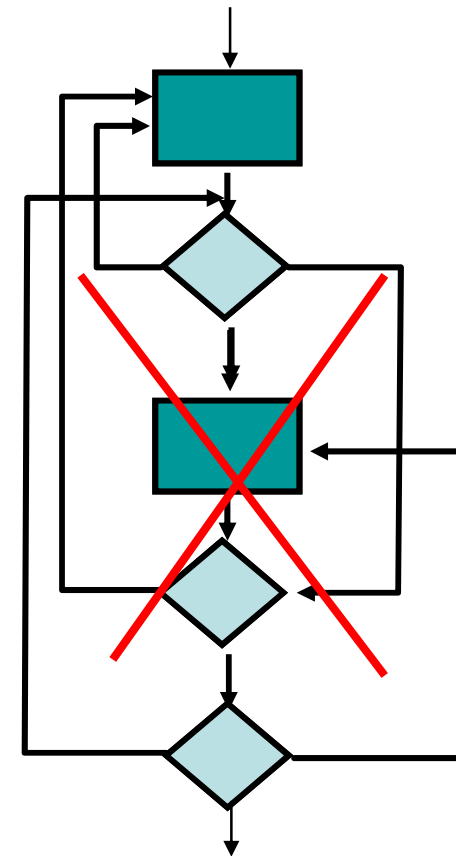
$n+1$



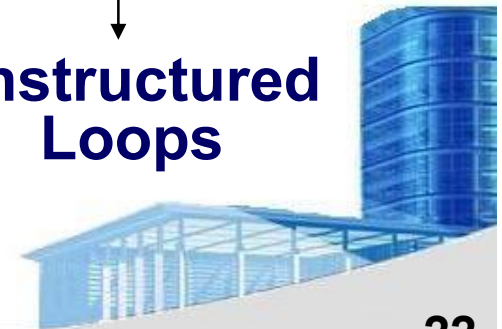
**Nested Loops**



**Concatenated Loops**



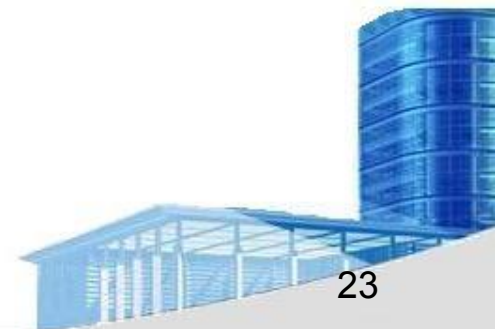
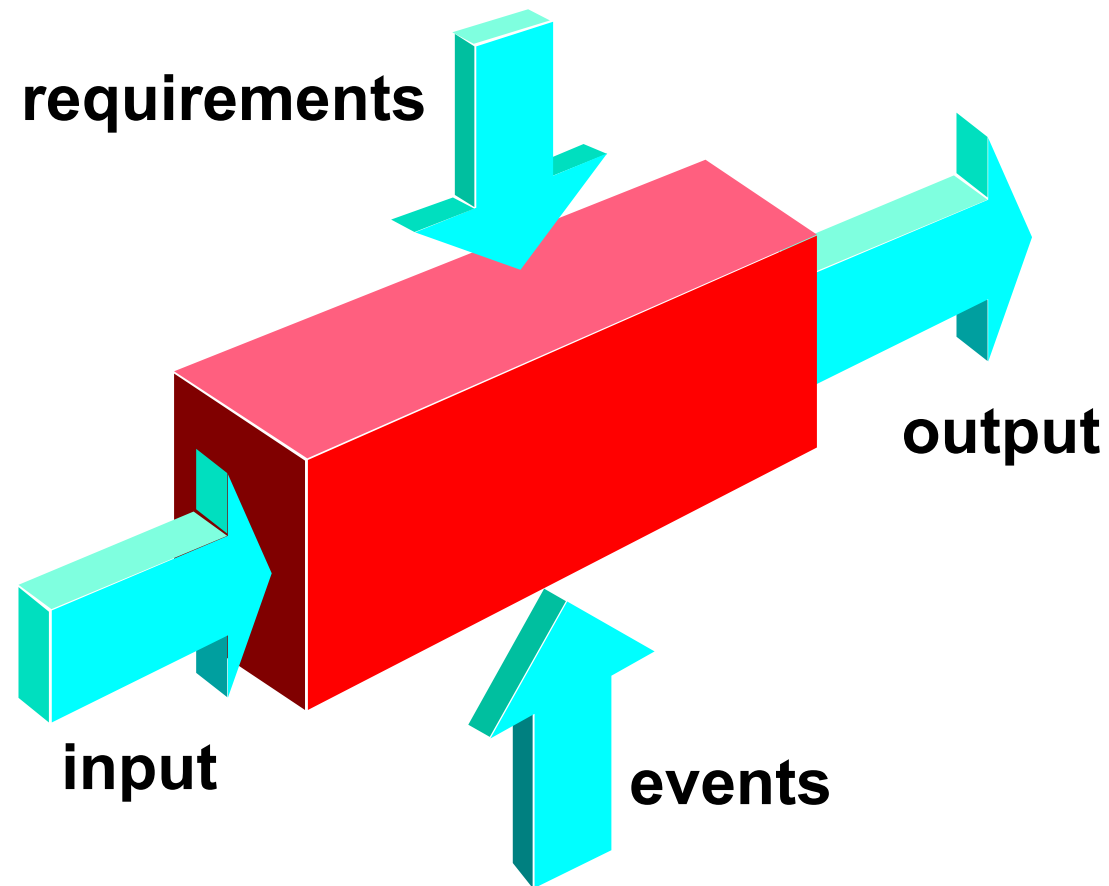
**Unstructured Loops**







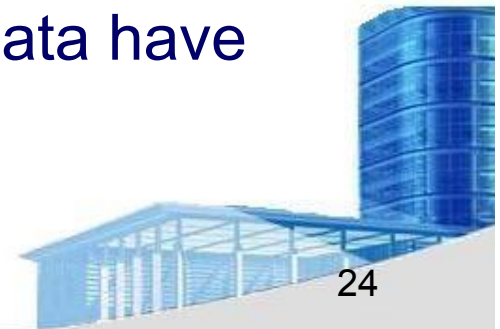
- **Black-Box Testing**







- **Black-Box Testing**

- How is **functional validity** tested?
- How is **system** behavior and **performance** tested?
- What **classes** of input will make good test cases?
- Is the system **particularly sensitive** to certain input values?
- How are the **boundaries** of a data class isolated?
- What **data rates and data volume** can the system **tolerate**?
- What effect will specific **combinations** of data have on system operation?





- **Black-box Testing Techniques**

- Graph-Based Testing
- Equivalence Partitioning 
- Boundary Value Analysis 
- Orthogonal Array Testing





# Black-box Testing

## ➤ Equivalence Partitioning

An ***equivalence class*** represents a set of valid or invalid states for input conditions, so that there is no particular reason to choose one element over another as a class representative.

### Guidelines

- If an input condition specifies a ***range***, one valid and two invalid equivalence classes are defined.
- If an input condition requires a specific ***value***, one valid and two invalid equivalence classes are defined.
- If an input condition specifies a member of a ***set***, one valid and one invalid equivalence classes are defined.
- If an input condition is ***Boolean***, one valid and one invalid equivalence classes are defined.



# Black-box Testing

## ➤ Boundary Value Analysis (BVA)

### Guidelines

- If input condition specifies a **range** bounded by values **a** and **b**, test cases should include a and b, values just above and just below a and b ( **Suppose  $a < b$**  ) .
- If an input condition specifies and **number of values**, test cases should be exercise the minimum and maximum numbers, as well as values just above and just below the minimum and maximum values.
- Apply guidelines 1 and 2 to **output conditions**, test cases should be designed to produce the minimum and maxim output reports.
- If internal program **data structures** have boundaries (e.g., size limitations), be certain to test the boundaries.



- **Model-Based Testing (MBT)**

- **---uses UML state diagrams**

- **Analyze** an existing behavioral model for the software or create one.
  - Recall that a behavioral model indicates how software will respond to external events or stimuli.
- **Traverse** the behavioral model and specify the inputs that will force the software to make the **transition** from state to state.
  - The inputs will trigger events that will cause the transition to occur.
- **Review** the behavioral model and note the expected outputs as the software makes the transition from state to state.
- **Execute** the test cases.
- **Compare** actual and expected results and take corrective action as required.
- **--- (Test-Case Design Web-sites: pp.515)**





- **Software Testing Patterns**

- Testing patterns are described in much the same way as design patterns (Chapter 12).
- Example:
  - Pattern name: **ScenarioTesting**
  - **Abstract:** Once unit and integration tests have been conducted, there is a need to determine whether the software will perform in a manner that satisfies users.
  - The **ScenarioTesting** pattern describes a technique for exercising the software from the user's point of view. A failure at this level indicates that the software has failed to meet a user visible requirement. [Kan01]







# 测试实例——载入影像

用例标识	Open_Req_Test		项目名称	遥感卫星项目		
模块名称	地面场景匹配模块		参考信息	《XX目标仿真模块》、 《用户手册》		
测试类型	功能测试					
测试方法	黑盒测试		测试日期	2014年4月2日		
用例描述	载入影像大小大于500M					
前置条件	无					
编号	测试项	基本操作	描述/输入/操作	期望结果	真实	备注
1	资源三号数据NAD级(1.12G)	点击【打开场景影像】选择测试数据	选择资源三号数据NAD级	数据图层树添加新影像数据节点，显示所包含的一个波段节点		
2	资源三号数据1级(572M)		选择资源三号数据1级	数据图层树添加新影像数据节点，显示所包含的四个波段节点		





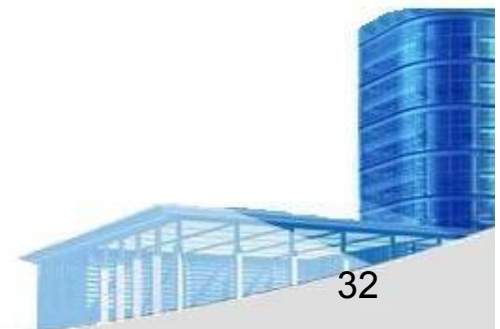
# Ch.24 Testing Object-Oriented Applications





- **OO Testing**

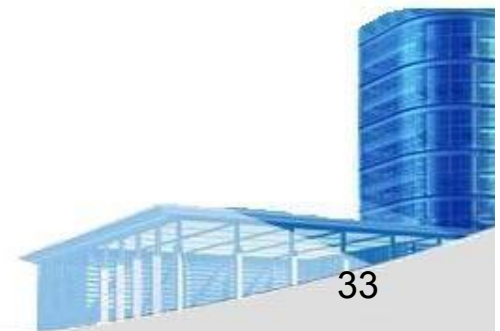
- To adequately test OO systems, three things must be done:
  1. the **definition of testing** must be broadened to include error discovery techniques applied to object-oriented analysis and design models
  2. the **strategy for unit and integration testing** must change significantly, and
  3. the **design of test cases** must account for the unique characteristics of OO software.





- ‘Testing’ OO Models

- The review of OO analysis and design models is especially useful because the same semantic constructs (e.g., **classes**, **attributes**, **operations**, **messages**) appear at the analysis, design, and code level
- Therefore, a problem in the definition of class attributes that is uncovered during analysis will circumvent (避免) side affects that might occur if the problem were not discovered until design or code (or even the next iteration of analysis).





- **Correctness of O-O Models**

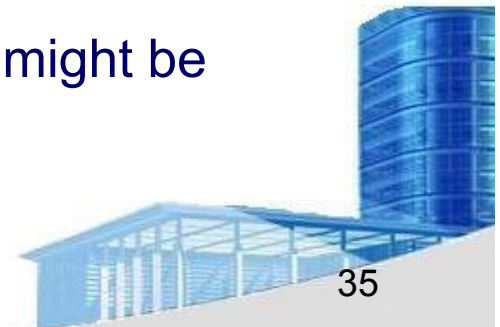
- During **analysis and design**, **semantic correctness** can be **assessed** based on the model's **conformance** to the real world problem domain.
- If the model **accurately reflects** the real world (to a level of detail that is appropriate to the stage of development at which the model is reviewed) then it is **semantically correct**.
- To **determine** whether the model does, in fact, reflect real world requirements, it should be presented to problem domain experts who will **examine** the class definitions and hierarchy for **omissions** and **ambiguity**.
- **Class relationships** (instance connections) are **evaluated** to determine whether they **accurately reflect** real-world object connections.





## • Class Model Consistency

1. Revisit the CRC model and the object-relationship model.
2. Inspect the description of each CRC index card to determine if a delegated(委派的) responsibility is part of the collaborator's definition.
3. Invert the connection to ensure that each collaborator that is asked for service is receiving requests from a reasonable source.
4. Using the inverted connections examined in the preceding step, determine whether other classes might be required or whether responsibilities are properly grouped among the classes.
5. Determine whether widely requested responsibilities might be combined into a single responsibility.





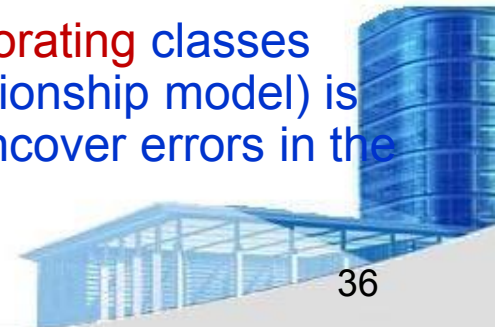
- **O-O Testing Strategies**

- Unit testing

- the concept of the unit changes
    - the **smallest testable unit** is the encapsulated **class**
    - a single operation can no longer be tested in isolation (the conventional view of unit testing) but rather, as part of a class

- Integration Testing

- **Thread-based testing** integrates the set of classes required to respond to one input or event for the system
    - **Use-based testing** begins the construction of the system by testing those classes (called independent classes) that use very few (if any) of **server classes**. After the **independent classes** are tested, the next layer of classes, called **dependent classes**
    - **Cluster testing** [McG94] defines a **cluster of collaborating classes** (determined by examining the CRC and object-relationship model) is exercised by designing test cases that attempt to uncover errors in the collaborations.







- **O-O T Methods**

- *Berard [Ber93] proposes the following approach:*
- Each test case should be uniquely identified and should be explicitly associated with the class to be tested,
- The purpose of the test should be stated,
- A list of testing steps should be developed for each test and should contain [BER94]:
  - a list of specified states for the object that is to be tested
  - a list of messages and operations that will be exercised as a consequence of the test
  - a list of exceptions that may occur as the object is tested
  - a list of external conditions (i.e., changes in the environment external to the software that must exist in order to properly conduct the test)
  - supplementary information that will aid in understanding or implementing the test.





- **Testing Methods**

- *Fault-based testing*

- The tester looks for **plausible faults** (i.e., aspects of the implementation of the system that may result in defects). To determine whether these faults exist, test cases are designed to exercise the design or code.

- *Class Testing and the Class Hierarchy*

- **Inheritance** does not **obviate** (避免) the need for thorough testing of all **derived classes**. In fact, it can actually **complicate** the testing process.

- *Scenario-Based Test Design*

- Scenario-based testing **concentrates** on what the **user does**, **not** what the **product does**. This means capturing the **tasks** (via use-cases) that the user has to perform, then applying them and their variants as tests.

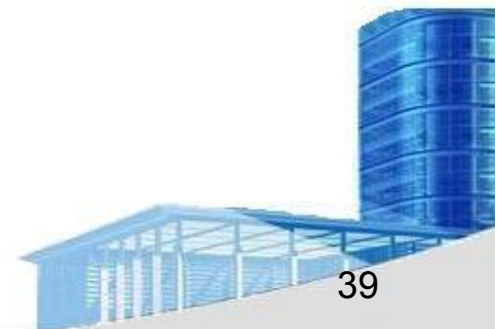




- **O-O T Methods: Random Testing**

- Random testing

- identify **operations** applicable to a class
- define **constraints** on their use
- identify a **minimum test** sequence
  - an operation sequence that defines the **minimum life history of the class** (object)
- generate a **variety of random (but valid)** test sequences
- exercise **other (more complex)** class instance life histories

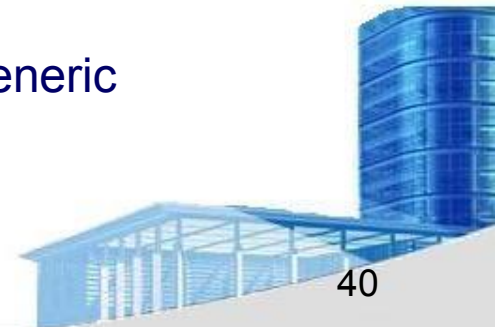




- **O-OT Methods: Partition Testing**

- Partition Testing

- reduces the number of test cases required to test a class in much the same way as equivalence partitioning for conventional software
- **state-based** partitioning
  - categorize and test operations based on their ability to change the state of a class
- **attribute-based** partitioning
  - categorize and **test operations based on the attributes** that they use
- **category-based** partitioning
  - categorize and test **operations** based on the generic function each performs

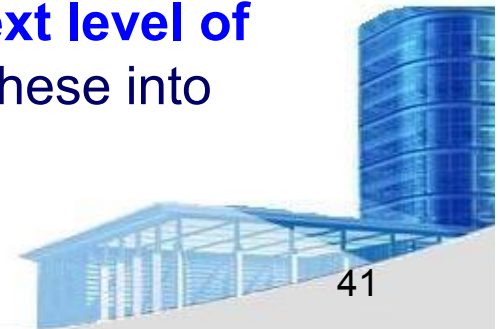




- **O-OT Methods: Inter-Class Testing**

- Inter-class testing

1. For each **client class**, use the list of class operators to generate a series of random test sequences. The **operators** will send **messages** to other **server** classes.
2. For each **message** that is generated, determine the **collaborator class** and the corresponding **operator** in the server object.
3. For each **operator** in the server object (that has been invoked by messages sent from the client object), determine the **messages** that it **transmits** (发送).
4. For each of the **messages**, determine the **next level of operators** that are invoked and incorporate these into the test sequence.





- **OOT Methods: Behavior Testing**

- The tests to be designed should achieve all **state** coverage [KIR94].

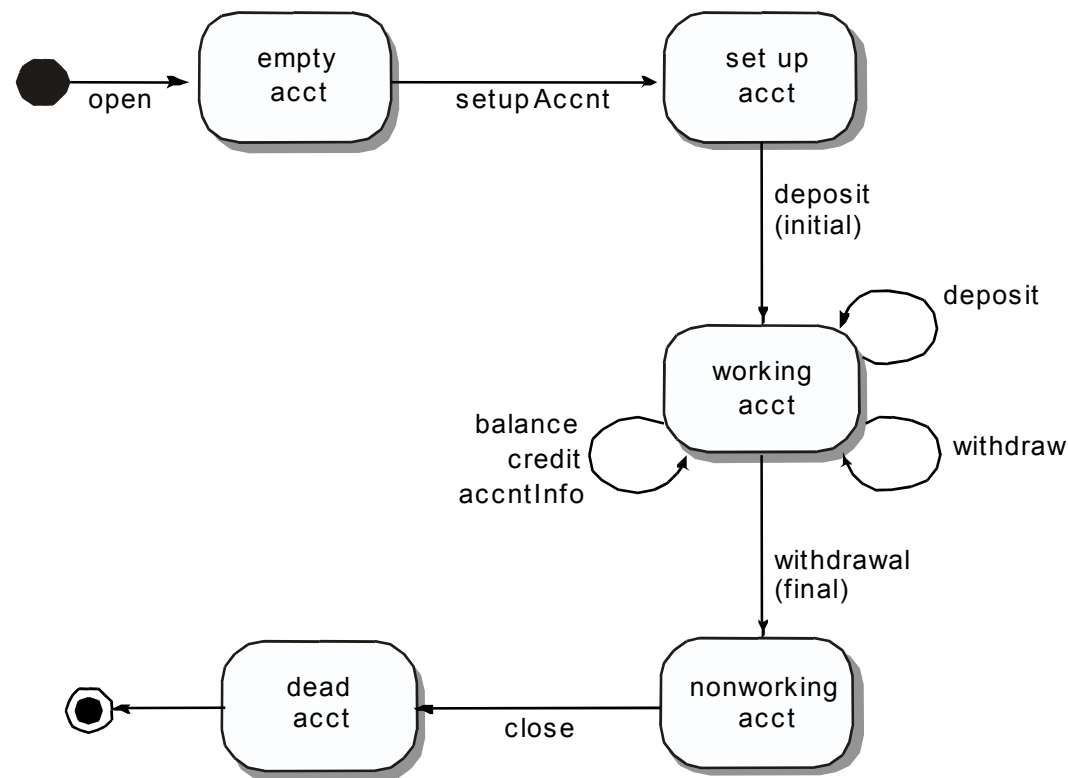
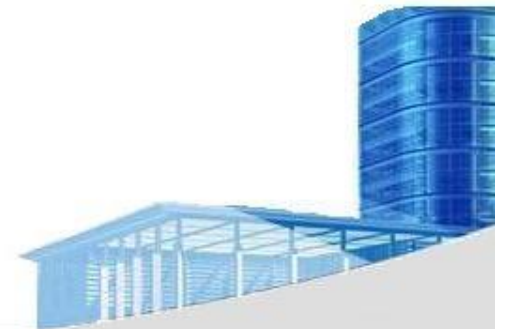


Figure 14.12 State diagram for **Account** class (adapted from [ KIR94])



# Tasks

- **Review** Ch. 23, 24
- **Finish** “Problems and points to ponder” in **Ch. 23, 24**
- **Preview** Ch. 25, 26
- Submit **Testing Plan** Due **June 8**
- Submit **v1.0** on **June 15**







## Quiz 2: AR (Augmented Reality) App

According to the videos presented, Please answer (either in Chinese or in English) following questions:

1. Which MobileApp form do you like most? Briefly describe it.
2. Can you describe the App's commercial developing steps?

Submit your answer(Including your ID and Name) to my QQ mailbox:  
[1281842242@qq.com](mailto:1281842242@qq.com) due 10:00pm, Wednesday (June 3)

