Testing the Architecture

Lesson Description

- This lesson discusses a variety of metrics that may be used to test the architecture, including coupling, cohesion, and stability. It also discusses other ways of testing the architecture including CRC cards and making the architecture executable.

Lesson Goal

- Participants will be able to test the goodness of their architecture using a variety of techniques and metrics.

Lesson Objectives

- Upon completion of the lesson, the participant will be able to:
  - Understand the steps of assessing architecturally significant use cases.
  - Test the architecture by making it executable
  - Test the architecture with CRC cards
  - Measure the architecture using the metrics for coupling, cohesion, and stability

Lesson Outline

- Testing the Architecture
  - Metrics to measure goodness
    - Coupling
    - Cohesion
    - Stability
  - CRC card session
  - Making the architecture executable
- Summary

Testing the Architecture

- After selecting one or more candidate architectures, you will test the choices to determine which is best
- The two primary ways of testing an architecture are :
  - Mathematical
  - Using requirements
- The tests can be applied by individuals or as part of a design review

Goodness of an Architecture

- A good architecture exhibits the same characteristics as a good object model
  - They are stable, easy to maintain, and flexible to change
  - This is good because most systems will be maintained for far longer than the time it took to develop them to begin with

Goodness of an Architecture

- Each subsystem should be strongly cohesive, loosely coupled, and stable in the face of change
- One of the most well known sets of metrics for OO classes is:
  - Chidamber, S. and C. F. Kemerer, “A Metrics Suite for Object Oriented Design”, *IEEE Transactions on Software Engineering*, 20, (6): 476-493, (June 1994).
  - http://www.pitt.edu/~ckemerer/CK%20research%20papers/MetricForOOD_ChidamberKemerer94.pdf

Goodness of an Architecture

- In this section we are going to look at older software engineering metrics, from the 70’s, dealing with good basic techniques
  - These are more appropriate at the architecture level where we are working with components and subsystems rather than individual classes

Coupling and Cohesion

- Two important concepts when evaluating an architecture are coupling and cohesion.
- Both are concepts for software engineering in general. They help to evaluate the design of modules.
- Coupling describes the relationship between modules.
- Cohesion describes the relationship within modules.

Coupling and Cohesion

- Coupling
  - refers to the extent to which one component uses another
  - should be minimal
- Cohesion
  - refers to the extent to which the actions of a component are tied together
  - should be maximal
- Summary ***"Low Coupling, High Cohesion"***

Coupling

- Coupling is determined by examining the number of dependencies (imports relationship) between subsystems
  - Dependencies limit reusability
    - A subsystem cannot be reused without reusing the subsystems on which it depends.
  - Strive for loose coupling (few connections) between subsystems

Types of Dependency

- There are two basic types of dependency:
  - Structural
    - A structural dependency between Packages indicates some type of static model association between the Classes in the two Packages
  - Usage
    - A usage dependency indicates that an operation in a Class in one Package has, as a variable, a member of a Class belonging to another Package.

Structural Dependency

- You find Structural Dependencies by examining the declarations of classes in a subsystem
- If a class in one subsystem references a class in another subsystem in one of these ways:
  - Sub Classing
  - Association
  - Attribute
- Then the two subsystems have a structural dependency

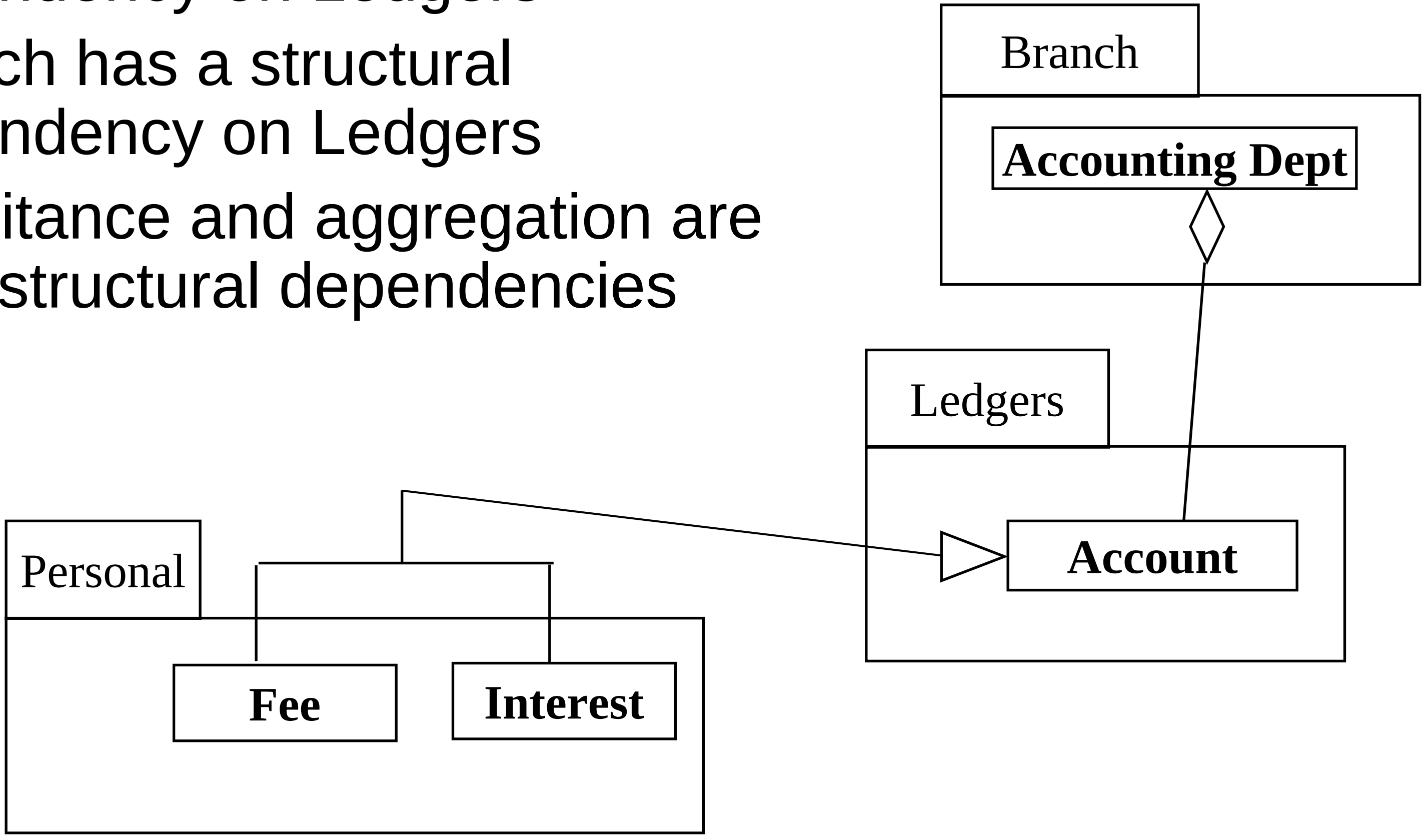
Structural Dependency Sample Code

```
public class Ledger
{
private Account
   MyAccount;
public void  Credit( ) { };
public void  Debit( ) { };
}
```

- Ledger has a structural dependency on Account
- If Ledger and Account are in different subsystems, those subsystems will have a structural dependency as well

Structural Dependency Diagram Example

- Personal has a structural dependency on Ledgers
- Branch has a structural dependency on Ledgers
- Inheritance and aggregation are both structural dependencies



Usage Dependencies

- You find Usage Dependencies by examining the operations of classes in a subsystem
- You have a usage dependency if a class in one subsystem has an operation which uses an instance of a class in another subsystem
- This instance can appear as:
  - An operation parameter
  - A return type
  - A local variable of the operation

Operation Parameter

```
public class Ledger {
public void  Credit(Account myaccount, money amount) { };
public void  Debit(Account myaccount, money amount) { };
}
```

- Account is passed as a parameter in Ledger’s operations

Return Type

```
Public class Ledger
{  public Account GetAccount (string key) { };
}
```

- Ledger knows about Account because it is a return type

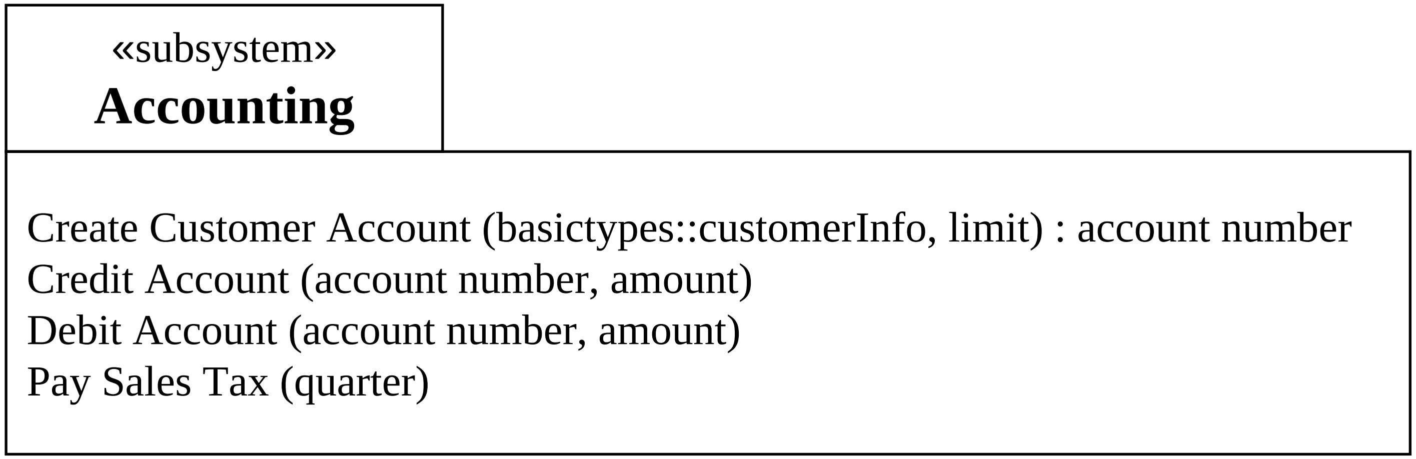
Local variable of the operation

```
public class Ledger
{
public Money GetBalance(string key) {
   Account  theAccount;
   theAccount = Database.GetAccount (key);
   return theAccount.GetBalance();}
}
```

- The Ledger creates a new Account object every time that the GetBalance( ) operation is called

Usage Dependencies and Subsystem Operations

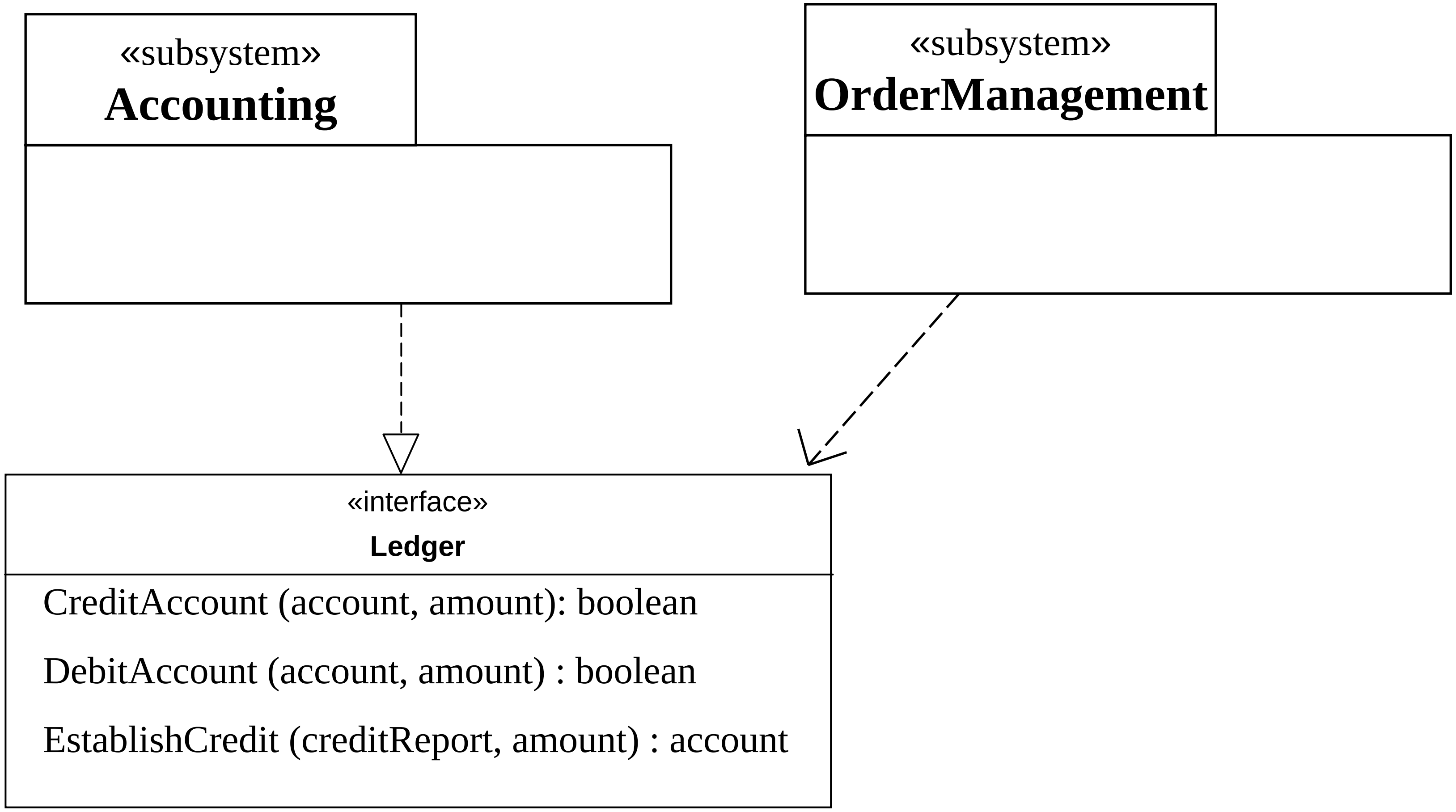
- If you do not yet have classes in the subsystems, then examine the subsystem operations for usage dependencies
  - Notice the parameter basictypes::customerInfo
  - This notation indicates that basictypes is the name of another subsystem


«subsystem»
Accounting
Create Customer Account (basictypes::customerInfo, limit) : account number
Credit Account (account number, amount)
Debit Account (account number, amount)
Pay Sales Tax (quarter)

Dependencies create coupling

- It didn’t matter whether we built a structural or a usage dependency; the Ledger could not be reused without the Account
- The dependency between ledger and account is a coupling between them, and therefore a coupling between their subsystems
- If there is only one coupling between the subsystems, they are weakly coupled
  - the more relationships there are between subsystems, the stronger the coupling

Interfaces and coupling


«subsystem»
Accounting
«subsystem»
OrderManagement
«interface»
Ledger
CreditAccount (account, amount): boolean
DebitAccount (account, amount) : boolean
EstablishCredit (creditReport, amount) : account

Varieties of Coupling

- Coupling can arise for different reasons. Some reasons are acceptable, some are not. The following is a list from poor to good:
  - Internal Data Coupling
  - Global Data Coupling
  - Control Coupling
  - Parameter Coupling
  - Subclass Coupling

Varieties of Coupling

- **Internal Data Coupling**: One module manipulates local data of another module. Difficult program reasoning!
- **Global Data Coupling**: Two modules depend on a common global data structure. Also: Difficult program reasoning.
- **Control Coupling**: The order in which operations of one module are to be performed is controlled not by itself but by another module.

Varieties of Coupling

- **Parameter Coupling**: One modules uses services from another. In this case parameters are passed. This kind of coupling is clean and can be checked.
- **Subclass Coupling:** A child can be treated as if it were (an instance of) its parent. Whether it is good or bad design depends on the kind of subclassing.

Class Relationships

- The relationships between classes are ranked from weak to strong:
  - Generalization / Realization
  - Dependency
  - Association
  - Aggregation
  - Composition
- A good object-oriented design will use the weakest relationships possible.
  - This makes the system easier to modify and reduces the impact of changes to the system.

Cohesion

- COHESION is the degree to which the responsibilities of a single subsystem are functionally related
- A subsystem is said to be strongly cohesive if the elements in that unit exhibit a high degree of functional relatedness
  - This means that every element in the subsystem should be essential for that subsystem to achieve its purpose

3 properties of Cohesion

- Subsystems that are strongly (functionally) cohesive demonstrate three properties, in order of importance:
  - The elements within the Subsystem are closed against the same type of change
  - The elements within the Subsystem are reused together
  - The elements within the Subsystem share common functions

Common Closure

- The elements within the Subsystem are all subject to the same types of changes, and immune to other kinds of changes
  - You have to consider the kinds of changes you might want to make in your application
    - port to a new platform
    - change the database
    - add functionality
    - be able to customize for particular clients
- Changes that impact one Subsystem should not ripple through the other Subsystems

Common Reusability

- The Subsystem is reused as an entity
  - The elements within it are inseparable
- Reusing an element within the Subsystem will cause all of the elements in the Subsystem to be reused.

Common Function

- The elements within the Subsystem cooperate together to render some usable service(s) to other Subsystems

Varieties of Cohesion

- Like coupling, cohesion can arise for different reasons. Some reasons are acceptable, some are not. The following is a list from poor to good:
  - Coincidental Cohesion
  - Logical Cohesion
  - Temporal Cohesion
  - Communication Cohesion
  - Sequential Cohesion
  - Functional Cohesion
  - Data Cohesion

Varieties of Cohesion

- **Coincidental Cohesion**: Poor design. Often result of "partioning" of larger program. In OO: classes with unrelated methods.
- **Logical Cohesion:** Logical connection, but no data or control connection. Example: a library of mathematical functions (sine, cosine,..).
- **Temporal Cohesion:** Operations are to be performed at the same time. Example: initialization modules.

Varieties of Cohesion

- **Communication Cohesion:** Operations, data access the same device or data. Example: manager modules.
- **Sequential Cohesion:** Operations are to be performed in a certain order. Often to avoid control coupling which is even worse.
- **Functional Cohesion:** Operations contribute to one single function. Desirable kind of cohesion.
- **Data Cohesion:** Data abstraction. A module exports functions with which its internal data can be accessed.

Constantine's Criteria for Cohesion

- **Larry Constantine says:** Given a sentence that specifies a module:

1. If the sentence contains a comma or more than one verb, the module probably has sequential or communicational cohesion.
2. If it contains words such as "first", "then", "after" the module probably has sequential or temporal cohesion.
   Example: "Wait for the instant teller customer to insert a card, then prompt for the PIN."

Constantine's Criteria

3. If the predicate does not contain a single, specific object the module is probably logically cohesive.
   Example: "Edit all data."
4. If it contains words such as "initalize" or "cleanup" the module probably has temporal cohesion.

Stability

- Another important measure of a subsystem is its stability
  - This refers to the impact of change on a particular subsystem
  - One of the most important things to do when constructing an architecture is to create subsystems which encapsulate things that you expect to change
  - We need to minimize the impact of those changes on the rest of the system

Stability

- Stability measures help us find the parts of the architecture that are most sensitive to change
  - Then we can design those parts so they won’t have to change often
  - This reduces the impact of change on the system
- When evaluating subsystems for stability, we look at two features:
  - How many subsystems depend on it?
  - How many other subsystems does it depend on?

Responsible

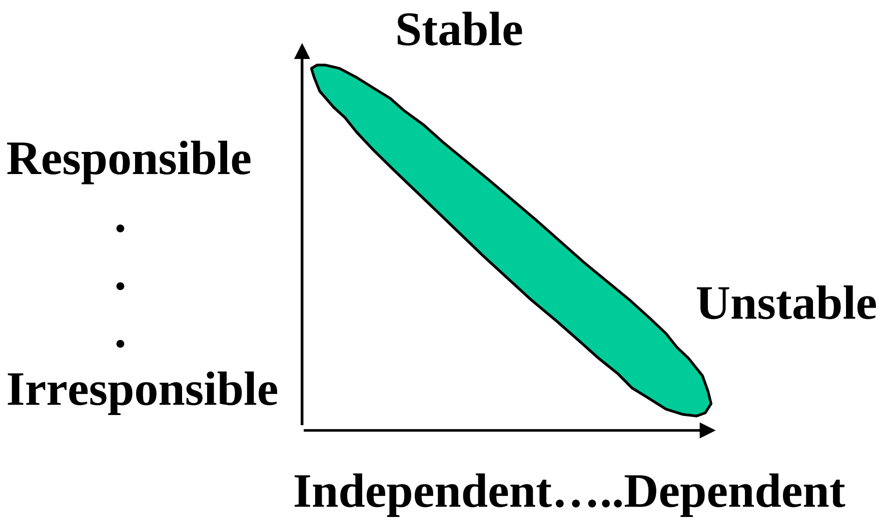
- Responsibility measures how many subsystems depend on a particular subsystem
  - Packages with many dependents are called responsible
  - Packages with no dependents are called irresponsible

Dependent

- Dependence measures how many other subsystems does a particular subsystem depend on
  - Packages with many dependencies are called dependent
  - Packages with few dependencies are called independent

The Stability Band

- One useful technique is to graph where your subsystem lies based on the its responsibility and dependency levels, as shown on the right
- Most subsystems will lie in the green band



Stable Subsystems

- Stable subsystems are both independent and responsible
  - Since so many subsystems depend on them it is difficult to change them without causing lots of other changes in the system
  - Since they are not dependent on other subsystems they are seldom changed.

Unstable Subsystems

- Unstable subsystems are both dependent and irresponsible
  - Since no subsystems depend on them, they can be changed without affecting the rest of the application
  - Since they depend on other subsystems, they will frequently have to change because of changes to the subsystems on which they depend.

Wyyzzk, Inc.

Design Tip

- Always make your dependencies in the direction of stability.
- Each subsystem should only depend on subsystems which are at least as stable as it is.

Goodness Metrics

- Now that we know what is good about an architecture, we will look at some basic metrics you can use on a particular architecture to measure its goodness. These are:
  - Relational Cohesion
  - Afferent Coupling
  - Efferent Coupling
  - Abstractness
  - Instability
  - Distance from the Main Sequence
  - Normalized distance from the Main Sequence

Relational Cohesion

- Cohesion inside a subsystem:

  $H = (R + 1) / N$

  R = Number of Relationships between Classes within the Subsystem

  N = Number of Classes within the Subsystem

  H closer to 0 shows low cohesion in the subsystem

  H around 1 is good cohesion

  H greater than 1 is strong cohesion, but the classes inside the subsystem may be too tightly coupled for a good design

Afferent Coupling Metric

- How strongly other subsystems are dependent on this subsystem

Ca = sum (Classes in other subsystems that depend on Classes within this subsystem)

- These dependencies can be structural (Association, aggregation or inheritance) or Usage
- The larger Ca is, the stronger the coupling

Efferent Coupling Metric

- How strongly this subsystem depends on other subsystems

Ce = sum (Classes in other subsystems upon which Classes within this subsystem are dependent)

- These dependencies can be structural (Association, aggregation or inheritance) or Usage
- The larger Ce is, the stronger the coupling

Strong coupling vs. type of coupling

- It is not just the number of couplings that we want to restrict; it is the kinds of coupling that need to be restricted.
- Abstract Subsystems should have low Ce and higher Ca
- Concrete Subsystems should have low Ca and higher Ce

Abstractness Metric

- How abstract is the subsystem

$$A = \frac{\text{\# of Abstract Classes in the Subsystem}}{\text{\# Classes in the Subsystem}}$$

- An abstract class is defined as any class that contains at least one pure virtual function

- A will vary from 0 to 1
- The closer to 1 A becomes, the more abstract the subsystem is
- The closer to 0 A becomes, the more concrete the subsystem is

Instability Metric

- How unstable is the subsystem?

  $I = Ce / (Ce + Ca)$

- I will vary from 0 to 1
- The closer to 1 I becomes, the less stable the subsystem is
- The closer to 0 I becomes, the more stable the subsystem is

Distance from Main Sequence Metric

- Abstract subsystems should also be stable, concrete subsystems should also be unstable

$$D = abs\,(A + I - 1)\,/\,sqrt(2)$$

- D ranges from 0 to ~0.7
- The closer D is to 0, the closer the subsystem matches the abstract vs. stability ideal

Normalized Distance from Main Sequence

- Same as previous, but normalized

  D’ = abs (A + I -1)

- D’ ranges from 0 to ~1
- The closer D’ is to 0, the better

Test Architecture with Requirements

- Besides the mathematical metrics, you may want to use your requirements to test your architecture
  - The metrics were used to measure correctness
  - Testing with use cases and other requirements measures completeness

Testing the Architecture

- We need to verify that the architecture we selected will support the application we are developing
- One technique you can use is a CRC card type session with your use cases (requirements) and subsystems

CRC cards

- CRC stands for: Class - Responsibility - Collaboration
- They were originally introduced by Kent Beck and Ward Cunningham in 1989
- They are a technique for assigning responsibilities and collaborations to classes or other entities

The CRC card itself

Class Name	
Responsibilities	**Collaborators**

CRC card details

- The CRC card is a 3x5 index card
- The name of the class goes on the top
- A vertical line separates the rest of the card into 2 parts
  - the left side is for responsibilities
  - the right side is for other classes which collaborate with this class to accomplish the responsibility on the left

A CRC card session to test an Architecture

- Take a stack of index cards and write the names of the subsystems you have already found across the top, one class per card
- Draw a line down the middle
  - labeling the sections is optional
- Hand out the cards to a group of people
  - engineers, business analysts, whoever is responsible for the requirements of the system

A CRC card session (cont.)

- Have a leader (who has no cards)
  - this person will read through each use case basic flow, step by step
- For each step, determine which subsystem is responsible for that behavior
  - write that behavior on the left side of the card for that subsystem
  - if another subsystem has to help out, write it’s name on the right side as a collaborator

A CRC card session (cont.)

- If the behavior does not go to any existing card create a new card for that behavior
  - you will frequently find new subsystems during a CRC card session
- If there is a disagreement about which subsystem should have a particular behavior, the subsystems may need to be redefined

Testing Other Scenarios

- After working with the basic flows, decide which alternatives are important or complex enough that you need to assign them to subsystems as well
- Go through the same process as you did with the basic flows
- Also, look at your non-functional requirements and assign them to subsystems
  - You will likely find that you need to create new subsystems to handle the non-functional requirements

Measure of success

- Things are going well if:
  - all the responsibilities for one subsystem fit on one 3x5 index card
    - If one card is not enough, the subsystem is too big and needs to be split
  - every card has some responsibility on it
    - If a card has no responsibilities, why do you have it?
      - perhaps some responsibilities need to be moved from other cards
      - perhaps this card is not needed

Measure of Success

- You must be able to allocate all of your use cases and requirements to subsystems in your architecture
  - you may need to add new subsystems to handle some of the behavior
    - how does this change your architecture?

Measure of Success

- If a set of subsystems work together to accomplish a use case or requirement, there must be communication paths (dependency relationships) between the subsystems
  - You should end up with an acyclic directed graph of all of your subsystems
- At the end of this exercise, you may decide to change your architecture to one of the alternatives you previously considered
  - Or you may decide that the architecture you picked works just fine

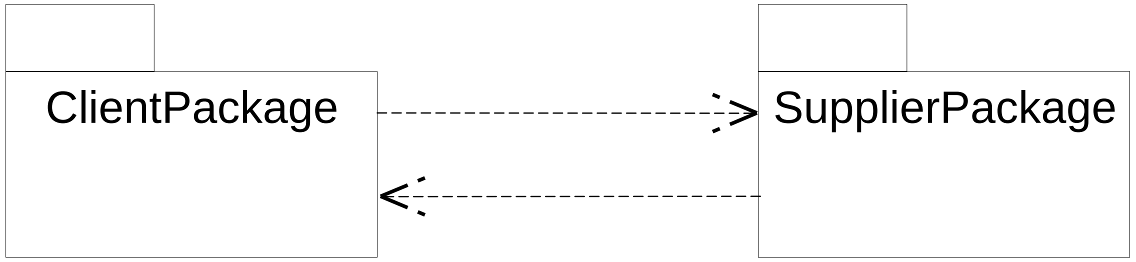
Avoiding Circular Imports

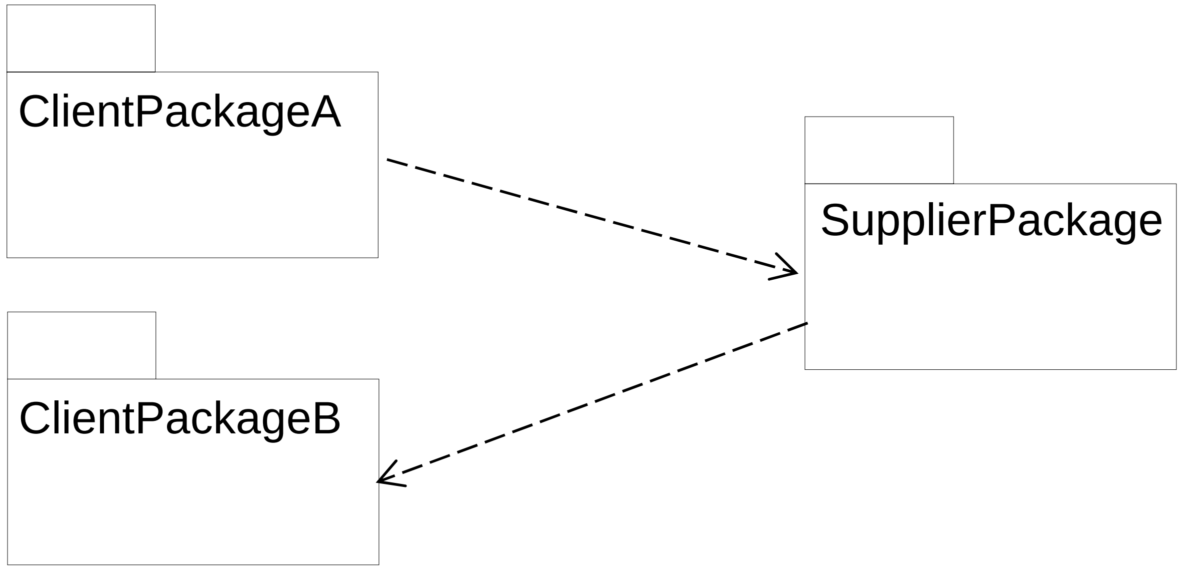
- It is desirable that the package hierarchy be acyclic
- This means that the following situation should be avoided (if possible)
  - Package A uses Package B which uses Package A
- Such a circular dependency means that Packages A and B will effectively have to be treated as a single package

Avoiding Circular Imports

- Circles wider than two packages must also be avoided
  - e.g., Package A uses Package B which uses Package C which uses Package A
- Circular dependencies may be able to be broken by splitting one of the packages into two smaller packages

Avoiding Circular Imports (cont.)





CRC using Static Architecture Diagrams

- The CRC card exercise can be done as described using index cards
  - The requirements or use cases for the subsystem are written on the index card
- Or you might do the same exercise using your static architecture diagrams
  - Update the diagrams as you go along by adding use cases, interfaces, or operations to your subsystems

Making the Architecture Executable

- Another way to test the architecture is to make an executable from the architecture and run it
- We need to determine how to convert the architecture into executable code
- To do this, identify the architecturally significant use case(s) and implement a thread which exercises all architectural layers
  - Architecturally significant use cases are those which determine what the architecture will be
  - Usually they are the important and complex use cases

Implementing Subsystems

- There are no subsystem type structures in Java
- But all we really care about are the implementation parts of the subsystem
  - The realization of a subsystem is the part that implements the subsystem operations, interfaces, and use cases (specification)
  - The realization of a subsystem is composed of classes and nested subsystems

Implementing Subsystems

- Start by allocating classes from the analysis model to the subsystems of the architecture
- These classes will be part of the realization of the subsystem
  - If the subsystem implements any interfaces, the operations in the interfaces must be implemented by the classes that realize the subsystem

Implementing Subsystems

  - If there are subsystem operations, those operations must be implemented by the classes that realize the subsystem
  - If the subsystem specification includes use cases, the use cases must be implemented by the classes that realize the subsystem

- As you allocate the operations and use case behavior to the classes in the subsystem, you will most likely add classes to the subsystem

Public Classes of a Subsystem

- Classes that can be called from outside the subsystem are public
  - The classes that implement subsystem interfaces and operations will be public classes
  - These classes are considered to be exported from the subsystem
- Some of your analysis classes will be in the public part of the subsystem
- You may add more classes to the public part to handle the interfaces, operations, and specification of the subsystem

Private Classes of a Subsystem

- Some classes will not be visible from outside the subsystem
  - These classes are part of the implementation of the subsystem, but not part of the interface
  - These classes are considered private to the subsystem
- Some of the analysis classes will be private, since they do not have operations corresponding to the specification of the subsystem
- You will add more classes to the private part of the subsystem as you continue with design

Simplifying Assumptions

- What we care about right now are the public classes of the subsystem
- Create the class headers for all the public classes of all the subsystems
- For now assume everything runs in one process on one computer

Simplifying Assumptions

- You can create simple implementations of the functions, to show the communication paths through the architecture
  - a function in one class calls a function in another class, which maybe just prints its name
- This will allow you to actually run some tests tracing paths through the architecture
- Remember you are not building your application, just putting together the framework of the architecture

Evaluate Results

- You have created an architectural proof-of-concept
- Now evaluate the Architectural Proof-of-Concept to determine whether the critical architectural requirements are feasible and can be met (by this or any other solution)

Application Framework

- Now that you have a working framework, you can add to it and modify it according to the needs of your application
  - For example, what if you decide to make the application multi-process?
    - Once you have decided which subsystems belong in which processes, you can change the simple function call interfaces to be inter-process communication channels
    - Now you can test just the inter-process communication part of your application
    - Once that works, the next step might be to put the processes on different computers and add CORBA

Summary

- We looked at a variety of ways of evaluating an architecture
- Mathematical metrics measure coupling, cohesion, and stability of a subsystem.
- Coupling refers to the connections between subsystems.
- Cohesion is the consistency within a subsystem.
- Stability measures the impact of change on a subsystem.

Summary

- We can also use the requirements to test the architecture.
- A CRC card session uses index cards for each subsystem.
  - A leader reads use cases and non-functional requirements, which are assigned to the various cards
  - This can also be done using an architecture diagram rather than index cards
- Alternatively, we can make the architecture executable and run tests to see if the architecturally significant use cases are handled.