12. Design for Testability

Reference Architectures and Patterns

Winter Semester 2008 / 2009

Prof. Dr. Bernhard Humm
Darmstadt University of Applied Sciences
Department of Computer Science

The lecture in the context of the entire course

1. Introduction
2. A reference architecture for business information systems
3. Application kernel
4. Persistence and transaction
5. Authorization
6. Client architecture
7. Exception handling
8. Business Intelligence
9. Systems integration
10. Service-oriented architecture
11. Design patterns and Refactoring
12. Design for testability

Agenda

➔ **Definition testability**

Architectural rules

Programming rules

Summary

Definition of Testability

- Ease of testing the functional and non-functional requirements of a system – initially as well as after systems modifications
- Accessibility of the system for tests:
  - Documentation of high quality
  - Composition in manageable components
  - Public interfaces
  - Production environment can be simulated in test environment

Source: Spillner, A.; Linz, T.: Basiswissen Softwaretest. dpunkt, 2005

Remark: The term „design for testability“ has ben branded in hardware testing. For testing integrated circuits ist is important to access internal test points from the outside

Criteria for testability

- Layering (e.g., user interface, application kernel, database access)
- Modularization: strong cohesion, lose coupling → localization of function → localization of test
- Limited special cases → high test coverage with few tests
  „Every if statement is one if statement too much for the tester“
- Errors are raised where they occur (not later)
- System can be parameterized (e.g., Date) by the test environment
- Easy debugging (localizing errors)

Agenda

Definition testability

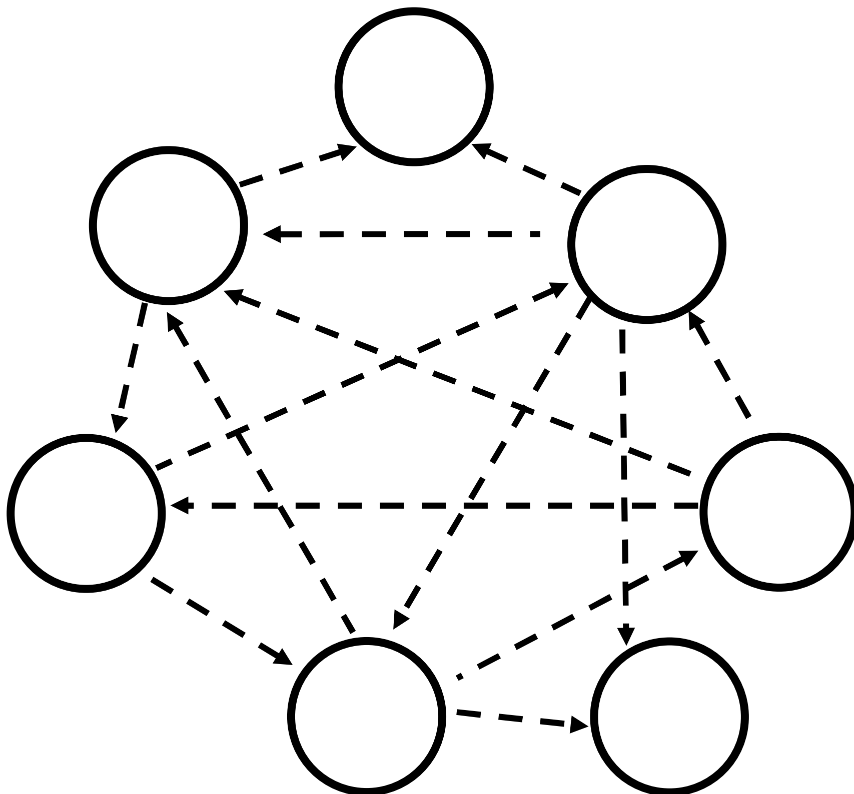
➔ **Architectural rules**

Programming rules

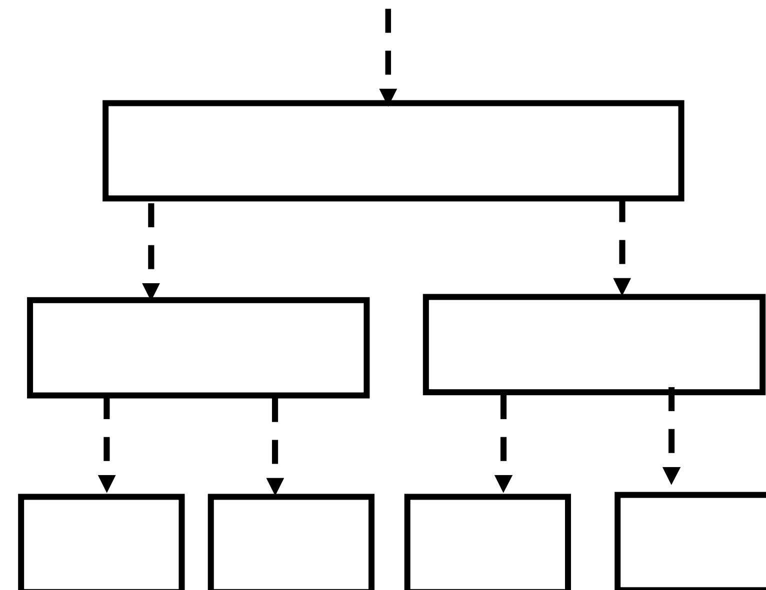
Summary

Layered architecture eases integration testing: reduction of dependencies to be tested

- Spaghetti architecture: many dependencies



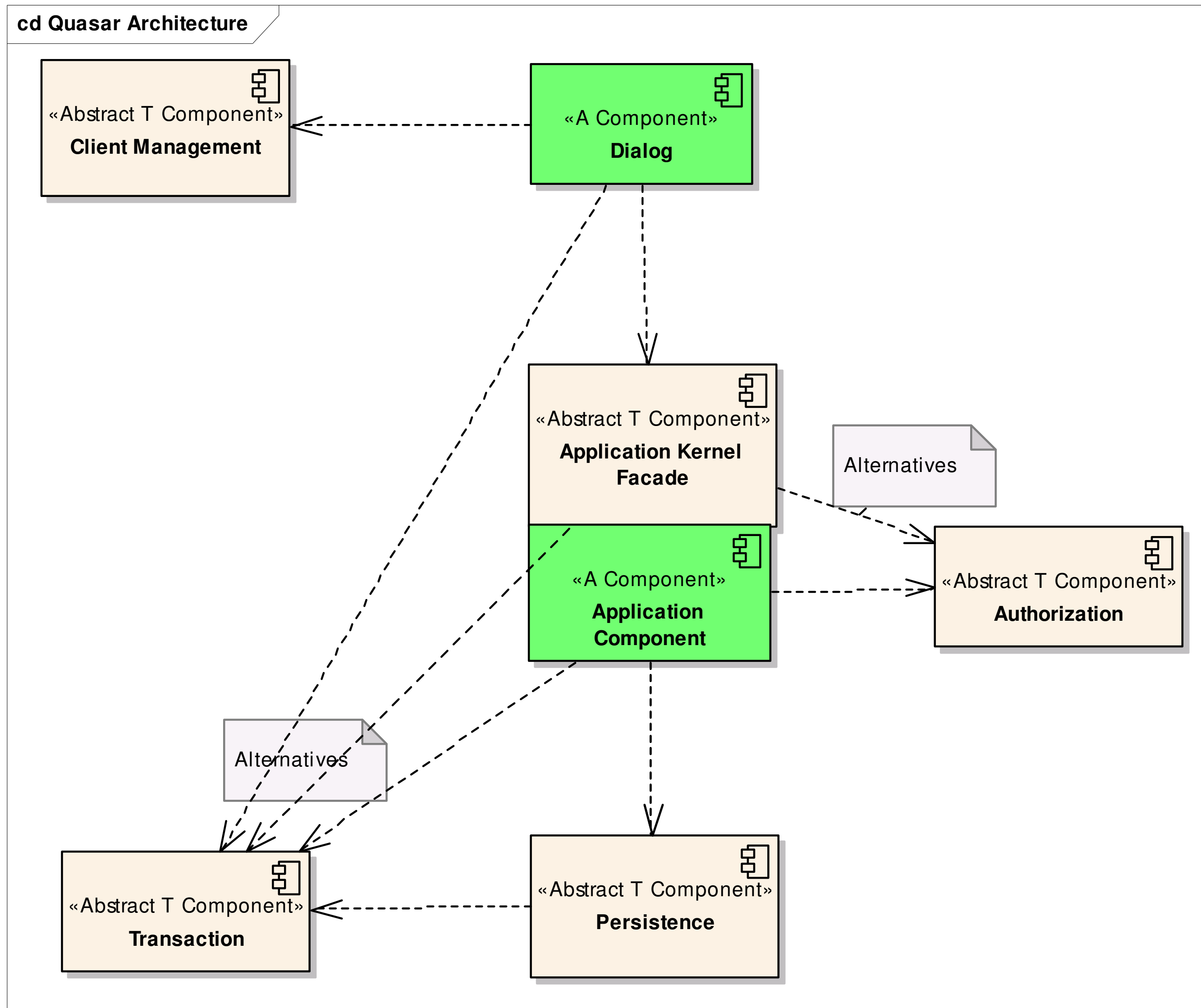
- Layered architecture: reduced number of dependencies



Three-Layer-Architecture eases regression testing

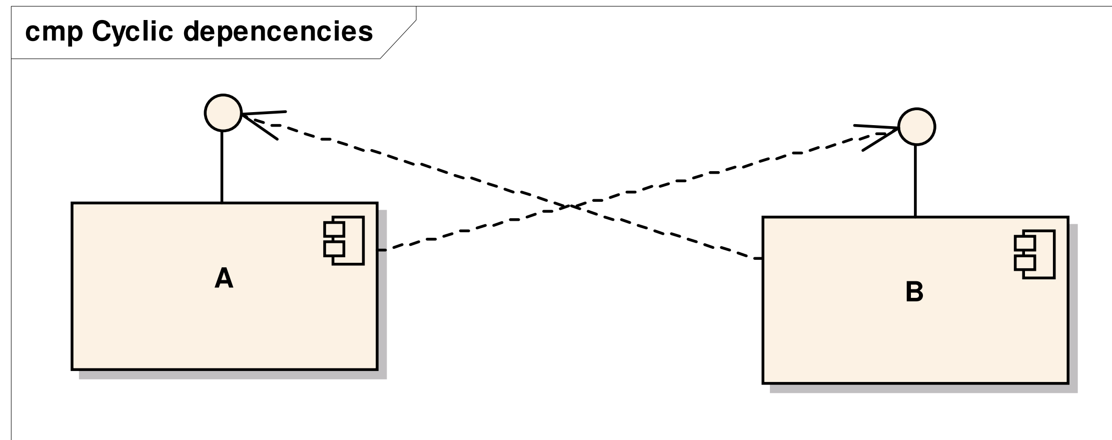
- Problems with testing GUIs:
  - Manual testing time consuming and error prone
  - GUI testing tools expensive
  - GUI modifications require manual re-testing or re-configuration of GUI testing tool (both expensive)
- Solution
  - Three-Layer Architecture: separation of GUI and application kernel
  - Regression testing of application kernel via interfaces (eg., JUnit)
  - Less critical GUI testing (Layout, navigation, help system etc.) can be done manually

Component orientation eases integration testing



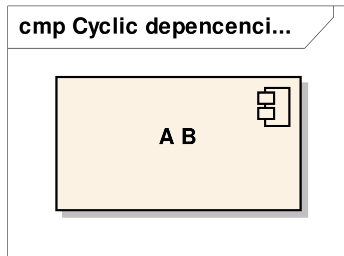
- Components = units of testing
- Component interface methods: subject to white box testing

Cyclic dependencies complicate testing

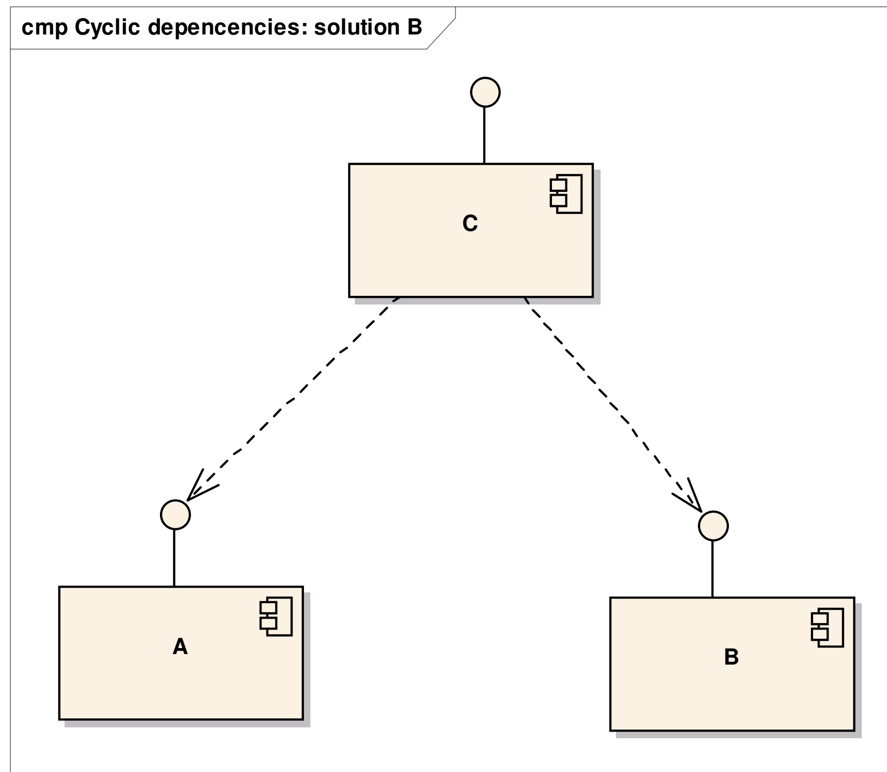


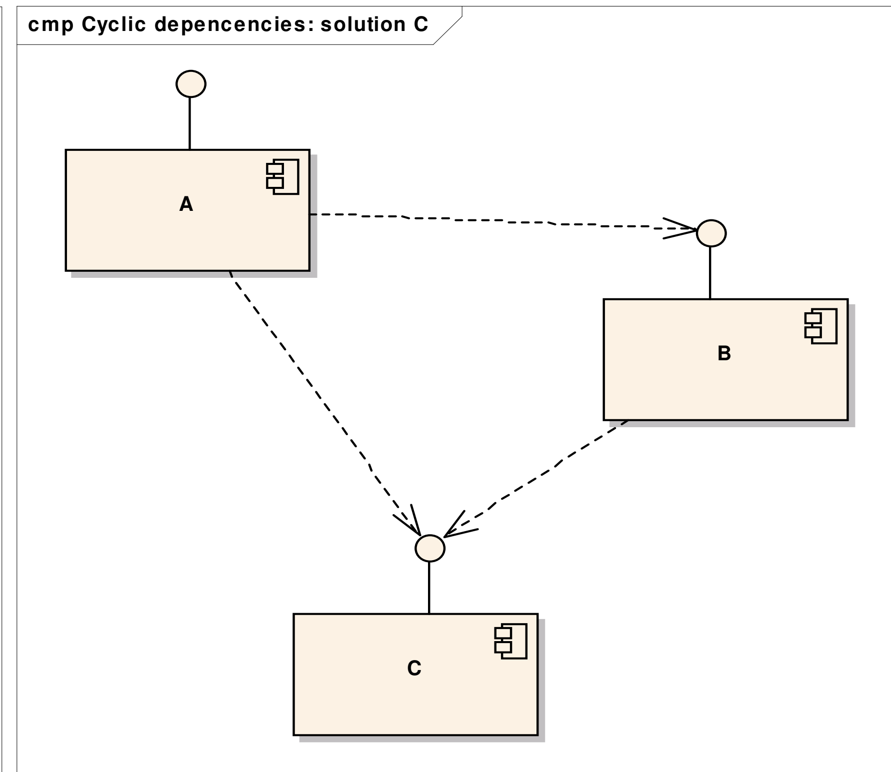
- Cyclic dependencies:
  - None of the components can be tested independently
  - When interfaces change both components have to be re-tested
- Acyclic dependencies:
  - Components can be developed and deployed independently
  - Components are tested along the dependency path
    (least dependent component first; most dependent component last)

Resolve cyclic dependencies: 4 methods A-C



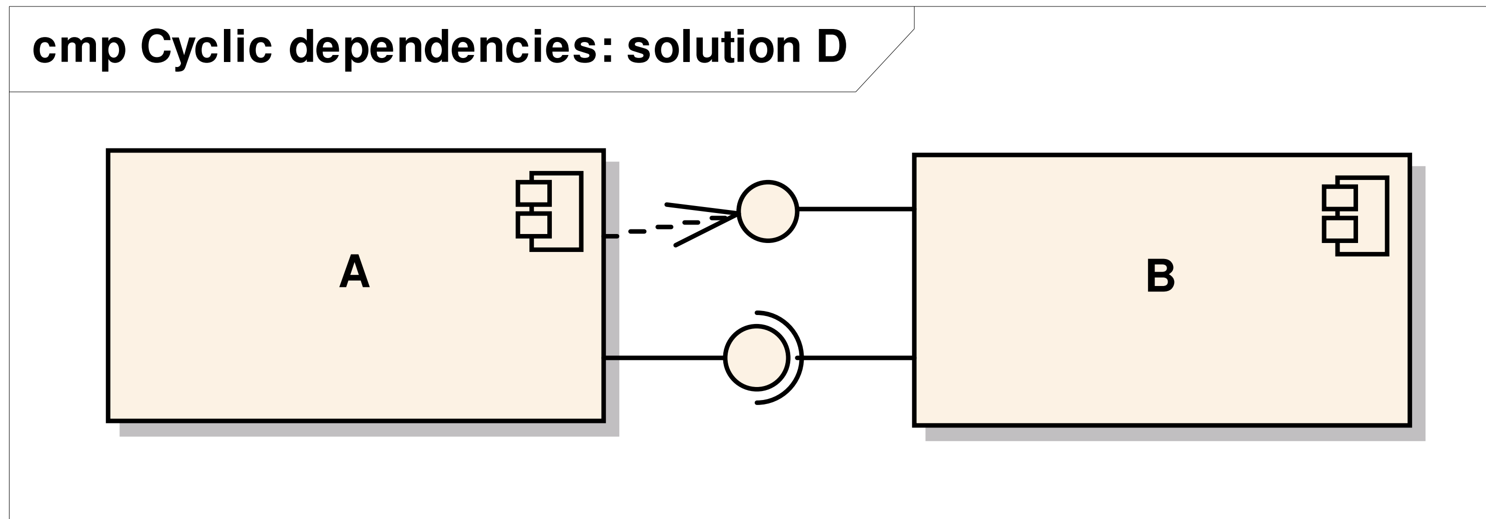
- A: combine components
- B, C: split components



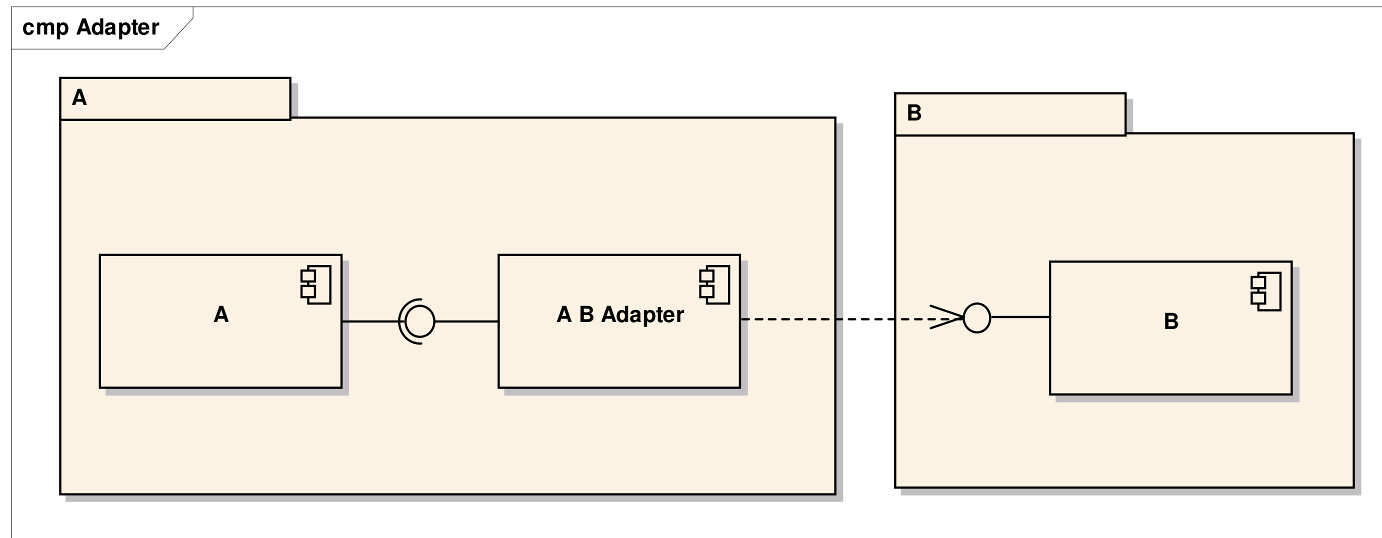


Resolve cyclic dependencies (cont'd)

- D: Dependency inversion via callback interface (observer / observable pattern)



Adapters ease integration testing



- Adapter pattern:
  - Component B specifies an interfaces for services it provides (implements)
  - Component A specifies an interface for services it requires – doesn‘t have to be identical with B‘s interface (maybe subset, different naming, etc.)
  - Adapter transforms A‘s required interface into B‘s provided interface
- Integration testing:
  - A can be tested with a dummy implementation of the adapter before B is delivered
  - Minor modifications of B‘s interface affect the adapter only.
    A doesn‘t have to be modified. A doesn‘t have to be re-tested

Agenda

Definition testability

Architectural rules

➔ **Programming rules**

Summary

Design for Testability – software categories

- Strong cohesion – lose coupling: → Reduction of dependencies to be tested
- Separation of software categories:
  - A-software: application logic
  - T-software: technical software
  - 0: software: neutral software (independent of technology and concrete application logic)
  - → improves software structure
  - → supports focussed testing

Design by Contract –
Errors are raised where they occur

Example: Bank transfer

```
void transfer(Account from, Account to, int amount) {
  assert from != null;
  assert to != null;
  assert amount > 0;
  assert (from.balance >= amount);
  …
}
```

Design by Contract – Advantages

- Robustness
  - Built-in testing – also during production
  - Avoiding even worse problems by detecting and handling errors early
  - Diagnosis of error easier → improved debugging
- Improved testability
  - Test for disallowed values does not have to be repeated in test cases
  - Identification of equivalence classes for testing easier

Programming conventions ease (white box) module testing

- High branch coverage C1:
  - No empty else blocks
  - No switch-statement without default
- Testing extreme values:
  - Define ranges for parameters and sizes of collections:
    - `#define PERS_NAME_LEN   88  // max. length`
    - `#define PERS_NBR_MAX   500  // max. size of drop down list`
- Reduced variants in white-box testing
  - Initialize variables

Configuration of date / time values for testing time-dependent functions

- Examples for time-dependent functions:
  - Insurance rates dependent on customer‘s age
  - End-of-term issues, e.g., billing, reminding, etc.
  - Interest rate calculation
- Problems:
  - System clock not configurable
  - Different system clocks (operating system, application server, database)
  - → Time-dependent functions not reproducible at arbitrary times
- Solution:
  - Central application-specific clock which can be configured by test environment

Logging / Tracing eases localization of errors

- Trace outputs ease localization of errors → error fixing simplified

Example Log4J output

```
15:32:00 main AWK:Workstep Geschaeftsfall.Gruppenauftrag;Ergebnis=ja ...
15:32:01 main Executer: { doTransition;State=pending;Event=Auftrag
15:32:01 main Executer: } doTransition;State=accepted;Result=OK;
```

Agenda

Definition testability

Architectural rules

Programming rules

➔ **Summary**

Summary

Proper design

- Improves understand ability of the system
- Eases integration
- Eases debugging and error localization
- Eases component testing, integration testing and systems testing
- Saves a lot of money in testing