SE 2AA4 Winter 2007

05 Software Design

William M. Farmer

Department of Computing and Software
McMaster University

13 March 2007

What is Software Design?

- Software design has two meanings:
  1. It is the activity that bridges the gap between requirements and implementation.
  2. It is the activity that gives structure to a software artifact.
- Software design is a special case of engineering design.
- The output of the software design activity is a software design in the form of a system decomposition into modules that specifies:
  1. The interface and implementation of each module.
  2. The relationship the modules have with each other.
- A software design is usually composed of several levels of abstraction.
  - The highest level is called the software architecture.
  - The lowest level is the code in a high-level programming language.

Design Strategies

1. Top down.
   - Module decomposition.
   - Refinement: A design is successively refined by adding details to it.
   - Transformation: A high-level design is transformed in a series of steps to a low-level design.
2. Bottom up.
   - Module composition.
3. Design for change.
4. Product families.
5. Little languages.

Design for Change: What Changes?

1. Algorithms.
2. Data representation.
3. Underlying abstract machine.
    - Hardware.
    - Operating system.
    - Programming languages.
    - Software libraries.
    - Database management systems (DBMSs).
4. Physical environment (peripheral devices).
5. Social environment.
6. Software development process.

Product Families

- A product family is a set of versions of a software product.
    - Some versions are successors of other versions.
    - Versions may differ from each other by the mix of services they provide.
    - Versions may differ from each other by the environments in which they are intended to be used.
- A product family should be designed as one system, not as a set of separate systems.
- Sequential completion is the wrong way to design a product family.
- A better approach is to:
    - Anticipate what family members will be needed.
    - Identify what structure is common to all members.
    - Delay decisions that differentiate members.

Modularization

- Questions:
  - What kind of structure does a modular system have?
  - What are the desirable properties of this structure?
- The structure can be viewed as various relations on the set of modules in the system:
  - Uses relation.
  - Is-component-of relation.
- The structure should include levels of abstraction:
  - Architectural design: What modules does a system need and how do they interact with each other?

    ⋮
  - Detailed design: What services does each individual module need to provide?

Uses Relation

- A module M_1 uses a module M_2 if M_1 requires a service provided by the interface of M_2.
  - The uses relation R is defined by

    $$M_1 \; R \; M_2 \text{ iff } M_1 \text{ uses } M_2.$$

  - In this situation, M_1 is the client and M_2 is the server.
- The uses relation is defined at design time.
- The user relation R determines a directed graph G called the uses graph.
  - G is defined by

    $$\text{There is an edge from } M_1 \text{ to } M_2 \text{ in G iff } M_1 \; R \; M_2.$$

Desirable Property 1

- The uses relation should be a hierarchy, i.e., the uses graph should be a directed acyclic graph (DAG).
- A hierarchy makes the software easy to understand, implement, and test.
- The presence of cycles in the uses graph implies that there is strong coupling between modules and that there is not a full separation of concerns.
- A system of modules with a hierarchical structure divides the system into levels of abstraction.
- A module M has level k in a hierarchy R is defined inductively by:
  1. Suppose that there is no module M' in R such that $M' \ R \ M$. Then M has level 0.
  2. Otherwise suppose k is the maximum level of all modules M' in R such that $M' \ R \ M$. Then M has level $k + 1$.

Desirable Property 2

- The fan-in of a module M in the uses graph G is the number of edges coming into M.
- The fan-out of a module M in the uses graph G is the number of edges going out of M.
- A good design tends to have high fan-in and low fan-out.

Is-component-of Relation

- A module M is composed of set S of submodules of M if M is the union of the modules in S.
  - S is a module decomposition of M.
- A module M' is a component of a module M if M' a member of a module decomposition of M.
  - The is-component-of relation R is defined by

$$M_1 \; R \; M_2 \text{ iff } M_1 \text{ is a component of } M_2.$$

- The is-component-of relation determines a hierarchy.
  - The modules that are leaf nodes in the hierarchy are physical modules composed of code.
  - The other modules are conceptual modules used to describe the physical modules in a hierarchical way.

Software Design Description Languages

1. Natural language descriptions.
2. Formal description languages.
   - Example: Textual Design Notation (TDN).
3. Graphical notations.
   - Example: Graphical Design Notation (GDN).
4. Modular programming languages.
   - Examples: Add, Modula-2, Oberon.
5. Object-oriented design languages.
   - Example: Unified Modeling Language (UML).

Parts of a Module Description

1. Name of the module.
2. Modules used (imported services).
3. Interface (exported services).
    - Names and signatures of components.
    - Descriptions of components.
4. Implementation.
    - Description of implementation.
    - Component modules.

Categories of Modules

- Library modules.
  - Package of related procedures.
  - Hide the algorithms of the procedures.
- Abstract objects.
  - Have state.
  - Hide a data structure.
- Abstract data types.
  - Export a data type.
  - Hides the representation of the type.
- Generic modules.
  - Parameterized by an abstract type which can be instantiated by a concrete type.
- Conservative and definitional extensions.
  - Have no access to the implementations of the modules they extend.

Anomalies and Exceptions

- An anomaly is an unexpected behavior by a service.
- An exemption is a signal to the client that an anomaly has been exhibited by a service.
  - An exception is thrown or raised when the anomaly occurs.
- A thrown exception is caught by an appropriate exception handler that tries to handle the exception.
- Ways an exception can be handled:
  1. An attempt is made to recover from the anomaly.
  2. The exception is thrown higher up the uses chain.
  3. The module's state is repaired as best as possible and then the service is allowed to fail.

Exceptions in Java (1)

- In Java there are three kinds of exceptions (which are objects of type `Throwable`):
    1. Checked exceptions of type `Exception` caused by internal anomalies that can be anticipated and recovered from.
    2. Runtime exceptions of type `Exception` caused by internal anomalies that usually cannot be anticipated or recovered from.
    3. Errors of type `Error` caused by external anomalies that cannot be anticipated or recovered from.
- Exceptions are thrown using a `throw` statement.

Exceptions in Java (2)

- The Catch or Specify Requirement requires code that might throw an exception to:
  1. Use a `try` statement to catch and handle the exception, or
  2. Use a `throws` clause to specify that the code's method can throw the exception.
- Only checked exceptions are subject to the Catch or Specify Requirement.
  - Runtime exceptions do not need to be checked.
- A `try` statement is composed of:
  1. A `try` block.
  2. A `catch` block.
  3. Possibly a `finally` block.