CMSC 735
A Quantitative Approach to Software Management and Engineering

Models and Measures (Product)
Guilherme H. Travassos
http://www.cos.ufrj.br/~ght

Outline

I INTRODUCTION
   I Products Measurement

I Part I: OO MODELS
   I Object-Oriented Paradigm
   I Engineering Object-Oriented Software
   I Representing OO Software Products using UML
      • High-Level Design
      • Low-Level Design

I Part II: OO MEASURES
   I Object-Oriented Software Measurement
      • High level design
      • Low level design
      • Coding
      • Testing
INTRODUCTION

Products Measurement

Metrics for any engineered product are governed by the unique characteristics of the product.

As for conventional software, OO metrics aim to:
- better understand the quality of the product
- assess the effectiveness of the process
- improve the quality of work performed

What makes an OO product different from a conventional one? How do we measure the quality of an OO system?

What characteristics of the design model can be assessed to determine whether the system will be easy to implement, amenable to test, simple to modify, and most important, acceptable to end users?
Products Measurement

Engineering Conventional Products

- the solution is function or data centered
- it uses algorithms, procedures (with different notations across the life-cycle) and data organization to describe the problem and organize information
- basically, modules arrange the software architecture
- information structuring technology (relational model, for instance) determines how the information will be stored and evaluated

Products Measurement

Engineering Conventional Systems:

- functional decomposition (structured development) and some possible metrics

<table>
<thead>
<tr>
<th>Specification</th>
<th>Design</th>
<th>Implementation</th>
<th>Testing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Function Points</td>
<td>Cohesion</td>
<td>Cyclomatic complexity</td>
<td>Coverage</td>
</tr>
<tr>
<td>pages of documentation</td>
<td>Coupling</td>
<td>Halstead's theory</td>
<td>Reliability</td>
</tr>
<tr>
<td>estimated lines of code</td>
<td>Complexity</td>
<td>lines of code</td>
<td>...</td>
</tr>
<tr>
<td>calendar time</td>
<td>Number of Modules</td>
<td>Data bindings</td>
<td>...</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>Span</td>
<td>...</td>
</tr>
</tbody>
</table>
Products Measurement

Object Oriented Paradigm:

- means to organize the software as a collection of discrete objects that incorporate both data structure and behavior
- characteristics:
  - identity
  - abstraction
  - classification
  - encapsulation
  - inheritance
  - polymorphism
  - persistence

CMSC 735
A Quantitative Approach to Software Management and Engineering

Models and Measures (Product)

<table>
<thead>
<tr>
<th>Part I: OO MODELS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Object-Oriented Paradigm</td>
</tr>
<tr>
<td>Engineering Object-Oriented Software</td>
</tr>
<tr>
<td>Representing OO Software Products using UML</td>
</tr>
</tbody>
</table>
  - High-Level Design |
  - Low-Level Design |
**Object-Oriented Paradigm**

**Identity**

Data is organized into discrete, distinguishable entities called objects.

An object has states and behaviors. A reference (handle) distinguishes different objects.

**Abstraction**

How many viewpoints (perspectives) can be used while building a system?

Abstractions form a hierarchy!
Object-Oriented Paradigm

Classification

Objects with the same features (attributes and behaviors) are grouped into a **class**

Each class defines a possibly infinite set of objects

an important OO development activity

depends on the abstraction level and the viewpoint (perspective) used by the developer
Object-Oriented Paradigm

Classification

Bicycle objects

Bicycle class
Attributes
frame size
wheel size
gears
material
Operations
shift
move
repair

Polygon objects

Polygon class
Attributes
vertices
border color
fill color
Operations
draw
erase
move

Object-Oriented Paradigm

Classification
Each object is said to be an instance of a class
Each object knows its class

Each instance keeps its own attributes value (state) but shares attribute names and behaviors (operations) with other instances of the class.
Object-Oriented Paradigm

Encapsulation

A class **encapsulates** attributes (names, types) and behaviors, **hiding** the object implementation details.

**Bicycle**
- frame size
- wheel size
- gears
- material
- shift()
- move()
- repair()

**Polygon**
- vertices
- border color
- fill color
- draw()
- erase()
- move()

Inheritance

The **sharing** of attributes and behaviors among classes based upon a **hierarchical** relationship.

A class can be defined broadly and then refined (specialized) into successively finer **subclasses**.

Each subclass incorporates or inherits all the properties of its **superclass(es)** and its own specialized properties.
**Object-Oriented Paradigm**

**Structure identification:**
class hierarchy using single inheritance

![Class Hierarchy Diagram]

**Polymorphism**

the same behavior may behave differently on different classes

a behavior (operation) is an action or transformation that an object performs or is subject to. Normally, these actions are taken as response for some stimulus (message)
**Object-Oriented Paradigm**

**Polymorphism**

A specific implementation of an operation for a certain class is called a **method**. An operator may have more than one method implementing it. An OOPL automatically elects the correct method to implement an operation based on the name of the operation, carried data (parameters) and object’s class being implemented (**method signature**).

Therefore, new classes can be added without changing existing code.

**Persistence**

An object existence transcends time and/or space. The object state and its behaviors will be saved across the time or space.
Engineering Object-Oriented Software

OO Software Process Features:
- works with the encapsulation of data and behavior into independent units (objects)
- uses the same set of semantic constructs to represent the system from the requirements to the application
- supports different software life-cycles
- stimulates reuse

In general, it deals with (but not using a totally sequential approach):
- requirements description (textual description and scenarios)
- high-Level design
- low-Level design
- coding
- testing

Engineering Object-Oriented Software

High Level Design:
- Classes/Objects identification
  - real world (problem domain) objects definition
  - attributes and behaviors detailing
- Classes/Objects interaction identification
  - association and aggregation relationships
  - probably some inheritance relationships

Low-Level Design
- inserting computational features into the models (user interface, tasks, persistence,...) and some class libraries details (uses a bottom-up approach)
- good time to deal with non-functional requirements (performance, security, etc.)
- classes contract preparation
- starting code
Engineering Object-Oriented Software

Coding
- translates OO models to some OO programming language
- refines hierarchical structures
- in some situations, makes some generalization for future use (components)

Testing
- unit testing
- integration testing
- system testing
- acceptance testing

Representations
- Static views:
  - objects
  - attributes
  - behavior
  - relationships: generalization, aggregation and inheritance
- Dynamic views:
  - communication
  - control/temporization
  - states
- Restrictions:
  - in the structure (attribute values, relationship cardinality)
  - in the dynamic behavior

Figure 1: O.O. Testing level
Representing OO Software Product using Unified Modeling Language - UML

UML can be used to
- specify, visualize, and document the design artifacts of a system under development.

OO Design Artifacts

- Dynamic View
  - use cases
  - activities
  - interaction
  - sequences
  - collaborations
  - state machines

- Static View
  - classes
  - relationships
  - associations
  - generalization
  - dependency
  - realization

- UML Diagrams
  - state machines
  - collaborations
  - packages
  - deployment

- Restrictions and Formalization
  - OCL - Object Constraint Language

UML is just a notational approach and does not propose/define how to organize tasks design. Therefore, it can be tailored to fit different development situations and software lifecycles.

3 – A gas station owner can use the system to control inventory. The system will either warn of low inventory or automatically order new parts and gas.

Requirements Description and Use Cases

High and Low Level Design

Coding and Testing
High-Level Design

UML Use Cases
- capture some participant-visible function
- can be used to represent system’s utilization contexts
- show some of the system requirements using high abstraction level

Diagram elements:
- actor: is a role that a participating plays to the system
- use-case: shows the visible system functionality
- extensions: extends a use-case
- uses: reuse of use-cases

High-Level Design

The Gas Station Use-Cases

- Customer
- CC System
- Printer system
- Owner
- Gas services
- Parking services
- Parts ordering system
- Controlling inventory
- accounting services
- Gas ordering system
High-Level Design

**UML Class diagram**
- Captures problem vocabulary
- Relates everything together
- Models information in a system
- Anchors behavior to classes
- Generates class structure declarations

**UML Relationships**
- Association
  - Aggregation: is the part-of relationship
  - Composition: the whole

**Example of a class diagram**
### High-Level Design

#### A Class Description

<table>
<thead>
<tr>
<th>Class name:</th>
<th>refuel</th>
</tr>
</thead>
<tbody>
<tr>
<td>Category:</td>
<td>Service</td>
</tr>
<tr>
<td>External Documents:</td>
<td></td>
</tr>
<tr>
<td>Export Control:</td>
<td>Public</td>
</tr>
<tr>
<td>Cardinality:</td>
<td>n</td>
</tr>
<tr>
<td>Superclasses:</td>
<td>services</td>
</tr>
<tr>
<td>Associations:</td>
<td>&lt;no rolename&gt; : gas in association &lt;unnamed&gt;</td>
</tr>
</tbody>
</table>

**Public Interface:**

| Operations: | price |

**Private Interface:**

| Attributes: | gallons  | price = 1.09 |

**Implementation:**

| Attributes: | gallons  | price = 1.09 |
| State machine: | No |
| Concurrency: | Sequential |
| Persistence: | Transient |

**Operation name:** price

Public member of: refuel

**Documentation:**

// Calculates the final price of the fuel

**Preconditions:**

gallons > 0

Object diagram: (Unspecified)

**Semantics:**

final_price = gallons * price

Object diagram: (Unspecified)

**Concurrency:** Sequential

### High-Level Design

**UML Package diagrams**

- used to break down a large system into smaller and meaningful sets of classes (clusters)
- shows the dependency among classes of different packages. A dependency exists between two elements if changes to the definition of one element may cause changes to the other (coupling)
  - Possible dependencies between two classes:
    - one class sends a message to another one
    - one class has another one as part of its data
    - one class mentions another one to be a parameter to an operation
- packaging is a vital technique for large projects:
  - allows keeping system’s dependencies to a minimum reducing coupling
  - is particularly useful for testing
High-Level Design

UML Package diagram

- **package**
  - Packaging Gas
  - Station classes

- **Products**
- **Service**
- **Customers**

High-Level Design

**UML Interaction diagram**
- used to describe how a functionality can be obtained from the objects
- typically, one interaction diagram represents just one use-case
- good to represent sequential functions, without much conditional or looping control
- two types of interaction diagrams:
  - sequence diagrams
  - collaboration diagrams
High-Level Design

Sequence diagrams

- An object is shown as a box at the top of a dashed vertical line (object's lifeline).
- An arrow between two objects' lifelines is called a **message** (labeled at least with the message name):
  - **self-delegation**: a message that an object sends to itself
  - **condition**: indicates when a message is sent
  - **iteration marker**: shows that a message is sent many times to multiple receiver objects
- Puts emphasis on the sequence of actions instead of the relationships between objects.

---

High-Level Design

Sequence diagram

- Parking Spot Request (account_number)
- Where to Park (available_parking_spot)
- Next Available (available_parking_spot)
- Lease Parking Spot (payment_time, payment_type, parking_spot)
- Authorize Payment (customer, amount, date)
- New Payment Request
- New Purchase (customer, parking_spot, date)
- Add to Bill (amount, date)
- New Payment Type Request
- Response Time => 30 secs or Credit Card Not Authorized and Payment Time = Now
- Response Time < 30 secs

---

services
High-Level Design

Collaboration diagrams
- objects are shown as icons
- arrows from the sender to receiver show the messages sent
- a number indicates the message order
- emphasizes which objects are statically connected

1: operation name (or an event)
2: operation name (parameters list)
Object: class name
attributes = values

2: operation name
(parameters list)

High-Level Design

Collaboration diagram

Customer
1: Parking()
3: parking_at(place)
4: new_purchase(customer,
parking_date, place)

GasStation
2: next_available()

Parking Spot

Purchase

Parking

Gas Station
Parking use case
**High-Level Design**

**UML State diagrams**

- Show possible object states, events that induce the transition from one state to another and possible actions associated to these events.
- Represent the dynamic model view of a simple object, object’s hierarchy, or the whole system.
- Normally, are only prepared for classes whose instances are very dynamic.

**Notation**

- **state:**
  - Means cumulative results of the object behavior/system functionality.
- **transition (event):**
  - Some occurrence which can induce an object/system state change.
- **state actions:**
  - Actions can be associated to states.
- **conditional transition:**
  - Some transition will be occur if a condition can be satisfied.
- **nesting states:**
  - States can be partitioned.

**Diagram Example**

- **State Diagram**
  - **Authorizing** to **Authorized**
  - **Authorized** to **Purchased**
  - **Rejected**

- **Conditions:**
  - `time <= 7`
  - `payment ok`
High-Level Design

Gas Station states diagrams

- inventory
- order_gas()
- order_part()

- low gas
- normal stock
- parts low in stock
- ordering gas
- ordering parts

- place : Strings
- is_available()
- next_available()

- available
- not available

- customer_request
- customer_release

UML Activities diagram

- can be used to model procedures activity flow
- an activity is similar to a state
- an event is similar to a transition
- decisions can be represented if an activity has more than one event
- normally, are associated to states diagrams of several classes

- activity A
- transition
- activity B

- initial activity
- activity X
- end

- activity B
- decision
- output A
- output B
- activity C
- activity D

The bar shows that one activity can broadcast data to other activities (parallels or not determined activities)
Low-Level Design

Advantages while using the same notation
- Developers don’t need to learn/use different constructs and representation
- Analysis/design transition is an entwining and soft process
- It improves reuse, extensibility and readability
- The results from the high-level design compose the initial problem domain design that should be modified and extended to satisfy all non-functional requirements

Problem domain design
- Model modification:
  - Early projects and coded classes reuse
  - Adjusting supported inheritance level
  - Performance and I/O issues
  - Data management support
  - Low level components
  - High level design extensions review
- Creating/Destroying Objects
  - Defines how each concrete class may create its objects (instantiation)
  - Describes how objects might be destroyed
  - It’s important to observe the restrictions applied to the class structure to maintain model consistency
- Inserting computational features: user interface, task and data management
- Decisions regarding deployment and physical software architecture
Low-Level Design

For instance, User interface design

UML Deployment diagram
- show physical relationships among system's software and hardware components
- useful to show components' system distribution
- basic notation:
  - a diagram node means some computing unit
  - a node connection shows some communication possibility
  - a component means a module (code)
  - components' dependencies are the same as the packages' dependencies
I Moving to code
   I A bare classification of OO PL’s:
      • Pure:
        – uses just OO constructs in its definition
        – enforces OO as the only approach to code
        – uses dynamic binding all the time
        – good for prototyping
        – normally, has tools to support OO coding, debugging and testing
        – Some examples: SMALLTALK, EIFFEL, and JAVA
      • Hybrid:
        – uses OO and non-OO constructs in its definition
        – doesn’t enforce OO as the only approach to code
        – eventually, uses virtual functions to implement dynamic binding
        – tools supporting coding, debugging and testing do not adopt OO
        – Some examples: C++, OO-COBOL, and OBJECT-PASCAL
Choosing a PL

<table>
<thead>
<tr>
<th>Features</th>
<th>Smalltalk</th>
<th>Eiffel</th>
<th>Java</th>
<th>C++</th>
<th>Object-Pascal</th>
</tr>
</thead>
<tbody>
<tr>
<td>Binding</td>
<td>dynamic</td>
<td>dynamic</td>
<td>dynamic</td>
<td>static</td>
<td>static</td>
</tr>
<tr>
<td>Overlay</td>
<td>allows</td>
<td>allows</td>
<td>allows</td>
<td>partially</td>
<td>partially</td>
</tr>
<tr>
<td>Garbage Collection</td>
<td>automatic</td>
<td>user</td>
<td>control</td>
<td>none</td>
<td>none</td>
</tr>
<tr>
<td>Inheritance</td>
<td>simple</td>
<td>multiple</td>
<td>simple</td>
<td>multiple</td>
<td>simple</td>
</tr>
<tr>
<td>Libraries</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>Productivity</td>
<td>high</td>
<td>high</td>
<td>high</td>
<td>medium/low</td>
<td>medium</td>
</tr>
<tr>
<td>Environment</td>
<td>nice</td>
<td>nice</td>
<td>nice</td>
<td>good</td>
<td>good</td>
</tr>
<tr>
<td>Tools</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>traditional</td>
<td>traditional</td>
</tr>
<tr>
<td>Learning</td>
<td>slow</td>
<td>nice</td>
<td>nice</td>
<td>good</td>
<td>good</td>
</tr>
<tr>
<td>Hardware dependency</td>
<td>low</td>
<td>high</td>
<td>high</td>
<td>medium</td>
<td>fast</td>
</tr>
<tr>
<td>User Interface</td>
<td>Self</td>
<td>Windows</td>
<td>XWindows</td>
<td>Windows</td>
<td>---</td>
</tr>
<tr>
<td>Enforces OO</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
<td>no</td>
</tr>
</tbody>
</table>

Moving to code

```cpp
class parking : public Services
{
    public:
        // Constructors (generated)
        parking();
        parking(const parking &right);
    // Destructor (generated)
    ~parking();
    // Get and Set Operations for Associations (generated)
    const parking_spot * get_the_parking_spot () const;
    void set_the_parking_spot (parking_spot * value);
    protected:
        // Additional Protected Declarations
        private:
            // Get and Set Operations for Class Attributes (generated)
            const float get_price () const;
            void set_price (float value);
            // Additional Private Declarations
            private: // Implementation
                // Data Members for Class Attributes
                float price;
                // Data Members for Associations
                parking_spot * the_parking_spot;
        };
    // Class parking
```