COMP 3700: Software Modeling and Design

(Design Verification and Validation)

Definition

- **Validation: Are we building the right thing?**
- **Verification: Are we building the thing right?**

Completeness and Consistency of UML Diagrams

- Syntactical errors: attributes should be named in a certain manner
- Completeness/omissions: missing type information for an attribute, or other missing information
- Inconsistency internally in a UML view, for instance in a class diagram
- Inconsistency across UML diagrams, for instance objects in a sequence diagram in relation to belonging class diagrams
- Design pattern violation: If a design is to be made according to a pattern, and errors according to the pattern structure are made

Simple Structural Checks

•Cardinality must be set for all associations This check looks through all associations and determines whether cardinality is set for both ends.

•A class can not have several similar 'opposite class' role names This check looks through all associations and determines whether any class has two or more identical role names at the ends of the opposite classes.

•At most one AssociationEnd in an association may be an aggregation or composition. In aggregations and compositions, only one of the association ends can be of the aggregation or composition type

•If an Association has three or more AssociationEnds, then no AssociationEnd may be an aggregation or composition Aggregations and compositions do not allow for ternary associations

Simple Structural Checks

•No Attributes may have the same name within a Class All attributes in a class are identified by name, and therefore the name must be unique.

•A realizing class for an interface must realize all the methods in the interface. When implementing an interface, the implementation class must contain all the methods of the interface

•Checking that operation names in sequence diagrams exist in class diagrams. For an operation to be valid in a sequence diagram, it must be a defined operation in the class that it is called from

•Find unused design artifacts This category contains only one check, which looks for elements present in a package that does not exist in a diagram. This is of interest since these items might be garbage unconsciously left behind.

Inter-consistency Checks for UML Diagrams

Sequence Diagram vs Class Diagram

Inputs:
1. A class diagram, possibly in several packages.
2. Sequence diagrams.

Goal: To verify that the class diagram for the system describes classes and their relationships in such a way that the behaviours specified in the sequence diagrams are correctly captured.

State Diagram vs Class Description

Inputs:
1. A set of class descriptions.
2. A set of state diagrams for the system objects.

Goal: To verify that the classes are defined, so that they can capture the functionality specified by the state diagram.

Inter-consistency Checks for UML Diagrams

Sequence Diagram vs State Diagram

Inputs:
1. A set of sequence diagrams.
2. A set of state diagrams for several objects.

Goal: To verify that every state transition for an object can be achieved by the messages sent and received by that object.

Sequence Diagram vs Use Case Diagram

Inputs:
1. A use case diagram for a part of the system, with its services.
2. One or more sequence diagrams for relevant system objects and services.
3. A set of associated class descriptions.

Goal: To verify that sequence diagrams describe an appropriate combination of objects and messages that capture the functionality from the use case.

Completeness and Consistency Analysis of UML Finite Statecharts

- **States and Transitions**
  - *Completeness* requires that in each basic state, for all possible events, there must be a transition defined.
  - *Completeness* requires that each state is targeted by (at least one) transition. Note that initial states have an incoming transition from the initial pseudo-state.
  - *Consistency* of the specification requires that in each state, only a single transition is triggered by a given event.
- **Guards**
  - *Completeness* requires that in each basic state, considering also inherited transitions, guards of transitions triggered by the same event form a tautology.
  - *Consistency* is checked by the following criterion: If there are two or more transitions that are originating from the same state and triggered by the same event, then their guards could not be true at the same time.

Evaluating a Class Design

- Evaluation is needed to accept, revise, or reject a class design:
  - Abstraction: Does it provide a useful one?
  - Responsibilities: Are they reasonable?
  - Interface: Is it clean and simple?
  - Usage: Does it provide the right set of methods?
  - Implementation: Reasonable?

Adequacy of Abstraction

- **Identity:** Are class and method purposes well-defined and related?
- **Clarity**: Can you give a brief, dictionary-style definition for the class purpose?
- **Uniformity**: Do operations have uniform level of abstraction.

Good or Bad?

- **Class Date:**
  - Date represents a specific instant in time, with millisecond precision.
- **Class Timezone:**
  - Timezone represents a timezone offset, and also figures out the daylight savings.

Adequacy of Responsibilities

- Clear: **Does class have specific responsibilities?**
- Limited: **Do responsibilities fit the abstraction?**
- Coherent: **Do responsibilities make sense as a whole?**
- Complete: **Does class capture the abstraction completely?**

```
class Complex {
private:
        double Real, Image;
public:
        Complex (double R = 0.0, I = 0.0)
        double getReal();
        double getImag();
        double setReal();
        double setImag();
        double Magnitude();
}
```

Adequacy of Interface

- **Naming**: Do names reflect the intended meaning?
- **Symmetry**: Are names and effects of pairs of inverse operations clear?
- **Flexibility**: Are methods adequately overloaded?
- **Convenience**: Are default values used when possible?

Poor Naming

```
class ItemList

private:

// …

public:

    void delete (Item item) // take item out and delete


    void remove (Item item) //take item out but do not delete


    void erase (Item item) // keep item in list with no information
```

Adequacy of Usage

- Consider the possible contexts that will use the object
  - Determine potential relevant and useful operations

```
class Location
private:
        int xCoord, yCoord /coordinates
public:
        Location (int x=0, int y=0)
        int xCoord(); //return Xcoord value
        int yCoord(); //return Ycoord value
};
```

```
//usage

Location point(100,100)
//shift point:

Point = Location(point.xCoord() + 5, point. Ycoord()
        + 10)
```

Revised Version of Location Class

```
class Location
private:
        int xCoord, yCoord /coordinates
public:
        Location (int x=0, int y=0)
        int xCoord(); //return Xcoord value
        int yCoord(); //return Ycoord value
        void ShiftBy (int dx, int dy)
};
```

```
//revised usage

Location point(100,100)

Point ShiftBy(5,10);
```

Implementation

Least important. One can easily change this aspect.

- **Poorly designs lead to low quality and problematic implementations.**
- **massaging a problematic implementation rarely produces effective improvement.**
- **Overly complex implementation indicates that**
  - **class is not well-conceived,**
  - **class has been given too many responsibilities.**