Design Verification and Validation

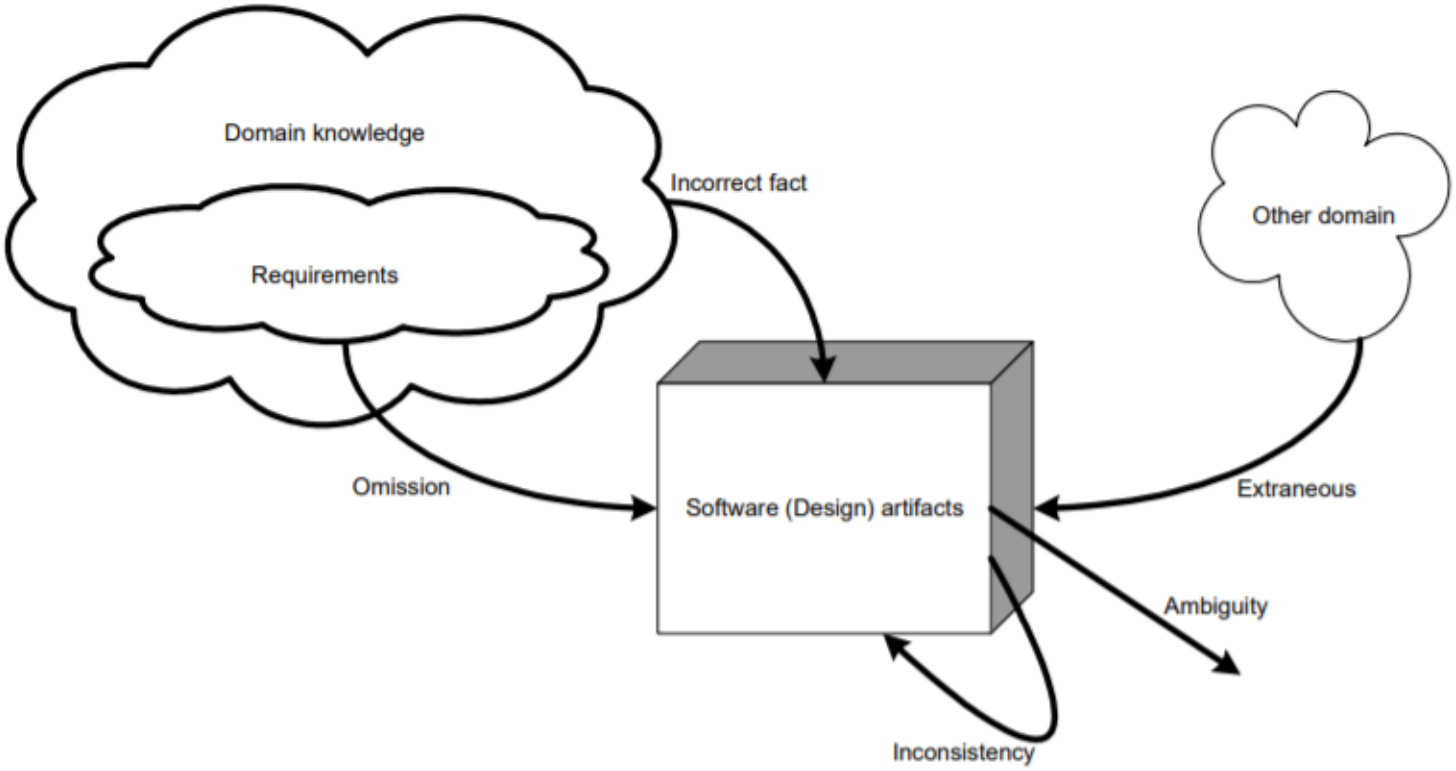
COMP 3700.002
Software Modeling and Design

Shehenaz Shaik

Overview

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

Defect Types



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
  - 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

- Associations
  - Cardinality must be set for all associations, for both ends.
  - A class cannot have several similar 'opposite class' role names
  - Association Ends (Roles) for one Association should not have the same name
  - At most one AssociationEnd may be an aggregation or composition
  - If an Association has >= 3 AssociationEnds, then no AssociationEnd may be an aggregation or composition

Simple Structural Checks (Contd.)

- Class checks
  - No Attributes may have the same name within a Class
- Interface checks
  - When implementing an interface, the implementation class must contain all the methods of the interface
- Sequence diagrams
  - Operation names in sequence diagrams must exist in class diagrams
- Attribute checks
  - Are there two attributes with the same name but different types?
  - The type of the parameters should be included in the Namespace of the Classifier
- Find unused design artifacts
  - Looks for elements present in a package that do not exist in a diagram. (These items might be garbage unconsciously left behind.)
- Parallel operations in subclasses
  - If subclasses of the same superclass have the same operation, perhaps the operation should be moved to the superclass

Inter-consistency Checks for UML Diagrams

- Sequence Diagram vs Class Diagram
- Inputs
  - A class diagram
  - Sequence diagrams.
- Verify
  - Class diagram describes classes and their relationships in such a way that the behaviors specified in the sequence diagrams are correctly captured.

Inter-consistency Checks for UML Diagrams (Contd.)

- State Diagram vs Class Description
- Inputs
  - A set of class descriptions.
  - A set of state diagrams for the system objects.
- Verify
  - Classes are defined, so that they can capture the functionality specified by the state diagram.

Inter-consistency Checks for UML Diagrams (Contd.)

- Sequence Diagram vs State Diagram
- Inputs
  - A set of sequence diagrams.
  - A set of state diagrams for several objects.
- Verify
  - Every state transition for an object can be achieved by the messages sent and received by that object.

Inter-consistency Checks for UML Diagrams (Contd.)

- Class Diagram vs Class Description
- Inputs
  - A class diagram (CDia), possibly in several packages.
  - A set of class descriptions (CDe).
- Verify
  - Detailed descriptions of classes contain all the information necessary according to the class diagram.
  - Descriptions of classes make semantic sense.

Inter-consistency Checks for UML Diagrams (Contd.)

- Class Description vs Requirement Description
- Inputs
  - A set of requirement descriptions (RD), mainly functional.
  - A set of class descriptions (CDe).
- Verify
  - Concepts and Services that are described by the functional requirements are captured by the class descriptions.

Inter-consistency Checks for UML Diagrams (Contd.)

- Sequence Diagram vs Use Case Diagram
- Inputs
  - A use case diagram (UC) for a part of the system, with its services.
  - One or more sequence diagrams for relevant system objects and services.
  - A set of associated class descriptions.
- Verify
  - Sequence diagrams describe an appropriate combination of objects and messages that capture the functionality from the use case.

Inter-consistency Checks for UML Diagrams (Contd.)

- State Diagram vs Requirement Description / Use Case Diagram
- Inputs
  - The set of all state diagrams (StD).
  - The set of all requirement descriptions (RD).
  - The set of use case diagrams (UC).
- Verify
  - state diagrams describe appropriate states of objects and events that trigger state changes as described by the requirements and use cases.

Completeness and Consistency Analysis of UML Finite Statecharts

- States and Transitions
  - Completeness
    - In each basic state, for all possible events, there must be a transition defined.
    - Each state is targeted by (at least one) transition.
      - Note that initial states have an incoming transition from the initial pseudo-state.
  - Consistency
    - In each state, only a single transition is triggered by a given event.

Completeness and Consistency Analysis of UML Finite Statecharts

- Guards
  - Completeness
    - In each basic state, considering also inherited transitions, guards of transitions triggered by the same event form a tautology.
  - Consistency
    - 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 for the type?
  - 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?

Examples

- 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
}
```

Hard to remember the difference

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
        )
```

Adequacy of Usage (Contd.)

- 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);
        //shift by relative coordinates
}
```

```
// Revised usage
Location point(100,100)

//shift point:
Point ShiftBy(5,10);
```

Implementation

- Least important. One can easily change this aspect.
  - Poorly engineered designs lead to low quality and problematic implementations.
  - Massaging a problematic implementation (without redesign) rarely produces any effective improvement.
  - It's only code… The issues here are primarily language syntax and semantics.
- Overly complex implementation indicates that
  - class is not well-conceived.
  - class has been given too many responsibilities.

Reading assignment

- References
  - Wiki:https://en.wikipedia.org/wiki/Software_verification_and_validation
  - UML Consistency Checks: https://www.idi.ntnu.no/grupper/su/sif8094-reports/2001/p8.pdf
  - https://www.researchgate.net/publication/2378462_Completeness_and_Consistency_Analysis_of_UML_Statechart_Specifications

Next sessions…

- Design Quality Evaluation