On OO Design Consistency in Iterative Development

Umesh Bellur,
School of Information Technology
I.I.T. Bombay, Mumbai, India – 400076
E-mail: umesh@it.iitb.ac.in

Vallieswaran V
School of Information Technology
I.I.T. Bombay, Mumbai, India – 400076
E-mail: vallies@it.iitb.ac.in

Abstract

In agile software development practices, the system evolves iteratively but does so in a manner consistent with its design rationale. Evolution often begins with last known design representation because it provides a high level view of the system that is easy to understand. However, the impact of a design change is poorly understood in terms of its effect on consistency of design. In fact, design evolution presents three important issues: consistency amongst design representations, traceability of a design change in code in order to maintain consistency and versioning of design entities along with versioning of code. In this effort, we propose a solution to the first two issues using a relational meta-model of various design & code entities and an algorithm to check consistency over this relational meta-model.

1. Introduction

Dictated by agility requirements, object-oriented systems development today is a highly iterative process. Every iteration of software needs to evolve the system in a manner consistent with the existing design captured through various design diagrams. These different design diagrams need to be consistent with one another and should be reflected consistently in the code for that design as well.

Producing a good design is often a tough task for novice programmers and so is evolving an existing system. To change the software, programmers start with its design representation because it provides high level of understanding of the system. However software developers at this stage either change code directly without updating the design or make a design change expressed as a change in one of the diagrams without having fully understood it’s impact. Hence we need methods and tools to analyze the impact of each design change made. Impact analysis should result in two outputs: a verification of the consistency of design and the impact of a design change on specific parts of the code. Design analysis for consistency is needed because working software can be represented only by consistent design and too often the code and design are out of sync with each other.

Design analysis for consistency is a complex and expensive process and needs efficient algorithms for automated analysis to be possible and useful. What makes the design analysis process complex? Design of a OO system is represented by different UML [1] diagrams. Each kind of UML diagram has different type of information about the system -class models represent structure, state models represent temporal behavior and so on. Each of these diagrams can be independently changed and hence we can be left with a set of versions of each design diagram. For us to check consistency, the entire set of design diagrams need to first be correlated with each other so that the right versions are being checked. We then need to trace through different design diagrams according to relation between the entities of design diagrams. Finally we need to verify the consistency rules for each relation. Typically programs (code) and designs are maintained separately. However for purposes of traceability, in our approach we consider code as just another design model. This way, our consistency checks also include code and ensure that a particular version of the code reflects the design properly and vice versa.

In this paper we propose a relational meta-model and a corresponding consistency checking algorithm. The problem is formally described with approach in section 2. The relational meta-model consists of both design and code entities, the details of which is discussed in Section 3. Consistency first looks at well-formedness of each individual design entity and then looks into relationships across entities in our meta-model. The set of rules for well-formedness of each design entity is discussed in Section 4. The consistency relations between design entities are described in Section 5. The results are presented in section 6. We present existing efforts in this area in Section 7. Section 8 concludes the paper with a report on current status and future work.

2. Problem formulation and approach

Currently UML is the de-facto standard for describing object-oriented design. UML has 7 diagrams to describe design - each diagram offering a different perspective.

The problems we see in using UML for documenting system design and using the same for design evolution are:

1. Each diagram can be independently edited. This can result in a problem with inconsistency of design since two views may not mean the same or in the worst case even contradict one another. Even if the UML models are consistent with each other, it does not imply that this consistency is reflected in code. We believe code
should be considered as one of the models so that the relationship between design entities and code representing it can be clearly defined. By doing so design to code correspondence can be maintained.

2. Most code generation tools in IDEs take into account class diagrams alone for code generation and this may leave the system in a state which is conformant with one of the design models but not the remaining ones.

A proactive view of consistency means we can force correlation between different model entities by prompting the user to change all the needed models on the change of any model to ensure that consistency is always maintained.

3. Relational meta-model

In this section, we discuss the relational meta-model that can be used to represent design entities. We believe that there are four different perspectives by which a system could be viewed in a design & development tool (IDE)¹:

1. Requirements View represents the requirements of the system. This view is represented by Use cases and Use case diagrams & is used by case tools [2].

2. Development View represents the design of the system. This view is represented by class diagrams, sequence diagrams and state diagrams & is used by modeling tools [2] and frame works.

3. Source View represents the source code of the system in any OO programming language. This view is represented by language syntax and used by development environments, compilers and interpreters.

4. Deployment View represents the deployment details of system. This view is represented by component diagram and deployment diagram & is used by configuration management tools [2].

In OO software development, the development view is directly related with all other views. Using the relation between the source view and development view, our second problem can be solved by using state, sequence and class diagrams for code generation. State diagrams can be converted to pre-conditions and post-conditions for each method of a class & sequence diagrams can be converted to sequence of operations of a method. Class diagrams can be converted to class template as is done by most IDEs today.

The first problem is solved by an algorithm to check consistency based on a relational meta-model combining all the four models of the system and relation between their entities. This relational meta-model is shown in figure 1.

Because of the redundancy of their information, we have chosen not to use activity & collaboration diagrams. The activity diagram only helps in creating method sequence diagrams but its other functionality is available in state diagram itself. Collaboration diagrams represent the information present in sequence diagram as operating context of objects rather than time based messages of interactions. So these two diagrams are not mandatory in our meta-model but users can use it for their reference.

The relational model entities are grouped to various views as described earlier. We now discuss each of these views in detail.

3.1 Requirements View

The requirements are captured as various use cases in this view. The requirements view consists of the use case diagram and use case sequence diagram (sequence diagrams per use case). These diagrams have design entities internal to them which are not represented in relational meta-model. Use case is the design entity in use case diagram which relates to the use case sequence diagrams so only these are represented separately in relational meta-model.

A Use case diagram typically contains one or more use cases. The collaborations needed for it’s realization are described by a use case sequence diagram. The use case sequence diagram is detailed by one or more method sequence diagrams which detail the execution of each

---

¹ Interactive Development Environment (IDE)
method or operation in terms of dependencies on other classes for realization of the method. The use case sequence diagram to method sequence diagram relation relates requirements view with development view.

3.2. Development View
The OO analysis and design process outputs are represented in development view which contains the class diagram, state diagrams and method sequence diagrams. A class and methods are present in all three diagrams and exchange information between each other and so are represented separately in our relational model.

The class diagram has one or more classes. Each class is associated with a state diagram and contains one or more methods. State diagram uses one or more method sequence diagram for state transitions. A method is represented by a method sequence chart. A method sequence chart has many messages (class + state + method sequence). The class is converted to class template in source. The state diagram is converted to pre and post condition of methods. The method sequence is converted to operations in method body. These are the relations that the development view has with the source view.

3.3. Source view
The code of the system is represented in the source view. Most OO languages have mechanisms to specify class, Pre and post conditions of method and method body. Class template of code is represented by class in the development model. Pre and post conditions of method ensure the state diagram properties. The method body impacts the method sequence diagram. This perspective is specific to programming language for e.g. in java state transition violations can be implemented by exceptions and report to user via exception handler.

The relations between source and development models are bi-directional since both are development entities one in programming language specific format and other in UML specific format. The UML notations of a state diagram and method sequences are used to link source code and design in ways similar to the use of the annotated reference language used in [3, 4].

3.4. Deployment view
The deployment view is used to generate builds of the system from source code & consists of deployment diagram and component diagram. The deployment diagram refers to component diagrams for the set of related components. The component diagram has one or more components and relations among them. The component in the component diagram refers to a class diagram in the development view. The class diagram describes the classes for the components. The classes describe the source for them. The build generated from deployment diagram gets the source from these relations and hence will be consistent with the design.

4. Design Consistency
As mentioned earlier design consistency comprises of consistency rules for individual design entities and consistency rules that trace relations across various design entities. We term the former notion termed well-formedness and is explained in detail in the rest of this section. The section following this discusses cross-entity consistency. In the rest of this paper, UML diagrams are described in XML Meta-data Interchange (XMI) representation for convenience. In XMI, every diagram is represented as XMI tree structure.

Each design entity representing a UML design model has a set of rules associated with it. Each rule is a boolean valued predicate on the features comprising that model entity alone. The individual features of design entities are currently not shown in the relational model since this is a conceptual model. For example: a use cases diagram consists of actors and use cases. In some cases, the rule can trace a relationship over to another model entity. These external dependencies are checked later as part of the cross-entity verification algorithm.

4.1 Well-formedness for Use Case diagram
A use case diagram displays the relationship among actors, system and use cases. The XMI tree for a use case diagram is shown below.

Use case diagram:
UML: Use_Case(defines)+

UML: SubSystem (Defines)+
  o UML: Use cases (Refer)+
    • UML: action sequence type: Use case (Refer)
    • UML: Association (Defines)+
      o UML: Use cases (Refer)
      • UML: Actor (Refer)
    • UML: generalization (Defines)+
      o UML: Use cases (Refer)
    • UML: Actor (Refer)
      o UML: Association (Defines)+
    • UML: Use case(Refer)

The well formedness rules on use case diagram are
  o Every Actor must use one or more use cases.
  o Every Use case must be used by one or more actors.
  o Every Use case must belong to system
  o Every Use case must reference a use case sequence diagram through (UML: action sequence type: use case).

These well formedness rules are checked over the XMI document by the tree structure verification using reference values that should be non-null. The last rule generates an external dependency that cannot be satisfied by the use case diagram alone.
4.2. Use Case Sequence diagram

A use case sequence diagram displays the time sequence of the objects participating in an interaction via messages. There is one diagram per use case. This XMI tree references method sequence diagrams corresponding to the methods invoked.

The XMI representation of a use case sequence diagram is shown above. The well formedness rules of use case sequence diagram are:

- For each variable of the general class used must be non-null and point to a valid class in the class diagram.
- The method sequence diagram reference must not be null for each method called and must point to a valid method sequence diagram.

4.3. Method Sequence Diagram

Method sequence diagrams display the time sequence of the objects participating interaction via messages. This consists of the vertical dimension (time) and horizontal dimension (different objects). Object is class instance with state. A Message is method call on an object.

The well-formedness rules are

- Every message must have a sender and a receiver object
- Every object must be the sender or the receiver in at least one interaction.
- Every object must reference a valid class and state diagram.
- Every message must be an instance of one class method for some class defined in the system.
- Every message must conform to the signature of method corresponding to the message.

4.4. Class Diagram

Class diagram models class structure and contents using design elements such as classes, attributes and methods. It also displays relationships such as inheritance, associations and aggregations.

The well-formedness rules for class diagram are

- Every Class diagram must have one or more classes
- Every association must have a source and target class
- Every class must have at least one attribute or method.

4.5. State Diagram

State diagram displays the sequences of states that an object goes through during its lifetime in response to received stimuli, together with its transitions and actions.

The well-formedness rules for a state diagram are

- Every State diagram must have one start and end state.
- Every State diagram must be related to one and only one class.
- State must be described by one or more attributes of the class.
- States should not have overlap of attribute values describing state.
- State change events must be done by messages corresponding to method sequence diagrams.
- Every state must be reachable from start
- From every state the end state should be reachable
- It must provide a non null reference to pre and post conditions for source code corresponding to that class.

4.6. Component Diagram

Component diagram displays the high level package structure and dependencies among components.

The well-formedness rules for component diagram are
o Component diagram must have one or more components
o Every inter-component relationship should have 2 terminating end classes (also called boundary classes) which are valid classes in the class diagram.

4.7. Deployment Diagram

Deployment diagram displays the configuration of system run-time processing elements and the software components, processes, and objects that live on them.

Deployment diagram:
- UML: System (Refers+)
  - UML: Component set (Refers+)

The well-formedness rules are:
- The deployment diagram must be related to one or more component diagrams
- Every component in the component diagram should be mapped to a physical system described in the deployment diagram.

These well-formedness rules are enforced by XMI tree structure. We examine XMI representation of diagrams for internal and external dependencies of diagram is stored them as comments in the XMI file itself. These comments will be used for examining consistency of entire system.

4.8. Source file Well-formedness rules

The source file format we consider currently is Java. Each class is coded in a different file. The structure of file is given below:

```java
public class Class_Name {
    <attributes list> (form class diagram)
    <method list> (form class diagram)
    <Method>{
        <Pre-Condition>(from state diagram)
        <method body>(from sequence diagram)
        <post-condition>(from state diagram)
    }
}
```

Every source file should follow this structure for verification purposes. The file should compile in Java without errors.

5. Cross-Entity Consistency

Here we present rules for checking consistency between various entities of the four views. The consistency rules are described such that if an entity X requires some other entity Y, the entity Y should be defined some where in the design, else the design is inconsistent due to X having an undefined reference Y. For every entity in relational meta-model there is set of defined features and referred features that are identified form the XMI representation of diagram. A referred feature is typically another entity or the feature of another entity in the model. The source supplies source entities for class, state and methods. For example, in the use case diagram, the actors and use cases are defined features while a use case sequence diagram is a referred feature.

Each design model i will have a defined set Di and a referred set Ri. So for a system with N design models, there will be ‘N’ defined set D1,D2,...,Dn and referred sets R1, R2, ... Rn. We will also have external definitions used that are imported to build the system.

The system is considered to be consistent if and only if any referred item is defined some where in the system. In other words the union of all the referred sets must be a proper subset of the union of all the defined sets. Mathematically:

$$U_i D_i \subseteq U_i R_i \quad \text{(1)}$$

This condition is both a necessary and sufficient condition for consistency. Some examples: If we don’t have state diagram for a class, then we can think of inserting a dummy state diagram with only one state ‘alive’ on which all methods of class except constructor is allowed & after every message it remains in the “alive” state except for destructor. This way the condition of a state diagram for each class is met in relational meta-model. Similarly if we don’t have sequence diagram for a method like a `get` method for some attribute, we can create a method sequence diagram with that object and no interactions.

6. Results

We have run our consistency checking tool over two medium sized applications 1. A Document viewer & 2. Automatic teller machine (ATM) simulation. The table below shows the results of our analysis.

<table>
<thead>
<tr>
<th>Table 1. Consistent project entity list</th>
</tr>
</thead>
<tbody>
<tr>
<td>Entities</td>
</tr>
<tr>
<td>-----------</td>
</tr>
<tr>
<td>Use cases diagram</td>
</tr>
<tr>
<td>Use cases</td>
</tr>
<tr>
<td>actors</td>
</tr>
<tr>
<td>Use case sequence</td>
</tr>
<tr>
<td>Methods</td>
</tr>
<tr>
<td>Method sequence</td>
</tr>
<tr>
<td>Dummy sequences *</td>
</tr>
<tr>
<td>Classes</td>
</tr>
<tr>
<td>State diagram</td>
</tr>
<tr>
<td>Dummy state diagram *</td>
</tr>
<tr>
<td>Class diagrams</td>
</tr>
<tr>
<td>Components</td>
</tr>
<tr>
<td>Component diagram</td>
</tr>
<tr>
<td>Deployment diagram</td>
</tr>
<tr>
<td>Source files (Java)</td>
</tr>
<tr>
<td>Total number of entities</td>
</tr>
</tbody>
</table>
In the above examples the design and code were not consistent. We found that dummy sequence diagrams and state diagrams are required to keep the design consistent according to the relational meta model. A dummy sequence diagram is for a method which does not have method calls while a dummy state diagram is for class which has no state values.

7. Related work.
In [6] T.Mens and his co-authors propose to use description logic to specify and detect inconsistencies between UML models. In description logic we have to specify rules for inconsistencies separately. So this will require different rule definitions for each diagram. In leverage’s the Object Constraint Language (OCL) to capture the evolution of UML models. Specific constraints describe how one model is different from another. Their technique formalizes the evolution of UML models, but it is unclear whether it could be extended to other modeling techniques. Their approach operates at a lower level of abstraction (i.e., classes, objects) and does not explicitly address the issue of versions, variants, and optionality. In their paper [8] the authors discuss the process of transforming UML representations to make it suitable for model checking. A model checker is an automatic tool that is able to compare the requirements and design descriptions of a given system. Karsten Diethers and Michael Huhn [7] present a tool for model checking UML designs. Their approach operates by converting UML to finite state automata, checking for errors in the automata and then reporting errors back in UML. The conversion rules are many and hard to change. this approach will not work with incomplete design or high level design. In [9] titled “Model Checking and Code Generation for UML State Machines and Collaborations” by Alexander Knapp and Stephan Merz, they discuss model checking and code generation from UML state and collaboration diagrams. The code generation component of HUGO produces Java code that behaves as prescribed by the state machines of a UML model. The Java code produced is a state pattern implementation of a single class with state diagram. In their paper [10] Lange and Chaudron propose a definition of completeness of a UML model and present a set of rules to assess model completeness. Model completeness can be decomposed into: well-formedness of each single diagram, consistency between diagrams, and completeness amongst diagrams. Here too they use a relational meta-model uses class diagram, message sequence chart, state diagram and use case diagram. However our work extends this significantly by including code artifacts as well as other deployment entities as part of the model.

8. Conclusion and Future work.
The relational meta-model we have proposed has views categorized according to different needs. Hence internal consistency checks can be customized without affecting the overall consistency checking mechanism. The consistency checking can also scale for large number of entities since the final check is based on union operations. As it stands, there is a need for getting the external definitions - we can eliminate this by performing omission of external references in well formedness rules.
Currently we have implemented these rules as a plug-in to Eclipse that we intend releasing to the open source community in the next few months.
The consistency checking rules can be extended to do change impact analysis where a particular change in any model can be turned into a set of instructions for changes to other parts of the design.

9. References