Software Engineering (CS3733) Notes

Christopher Myers

June 13, 2020

Contents

1 Requirements Gathering

A requirement is a condition or capability needed by the user. Types include functional (what should the system do? what abilities does it have) and nonfunctional requirements such as usability or performance. There are also constraints to a given project that are imposed by the client or the environment (e.g. a language requirement or a minimum-system-specs requirement)

Requirements are usually either conscious or unconscious. Conscious requirements are those that the clientis consciously aware of and has made clear, but there are unfortunately many unconscious requirements often. These can be requirements that are not known to be had, forgotten/unspken, or somehow glossed over (e.g. forgotten because an existing but to be replaced system already covers for it). Note a special subcategory of unconscious requirements that are "undreamed of" requirements, requirements that the client didn't know could exist.

Generally speaking, the customer wants their requirements to be met. If a spoken requirement is met, the customer is pleased. If it isn't met, they will be unhappy. For *un*spoken requirements on the other hand, the customer will be displeased if something they were expected is missing, pleased if it's present, and pleasantly surprised if an undreamed requirement is present.

Good requirements are correct, necessary, prioritized, unambiguous, conscise, and verifiable. Unfortunately, some of these can only be measured by the user (particularly the requirement on correctness), but for all other requirements there should be some way of verifying their completion.

There are a variety of techniques used to gather these requirements, including but not limited to interviews, surverys, brainstorming, task analysis & observations, and research. All this information is then combined with nonfunctional requirements and turned into a hopefully complete functional model. This functional model should involve user stories, epic stories, storyboards, use cases, and/or storyboards.

1.1 Brainstorming Sessions

Brainstorming sessions are *not* a whole lot of people sitting around a table throwing ideas out. The idea is to generate an enormous list of ideas, but under a no-criticism condition. These ideas don't even have to be feasible, the idea is just to get ideas out there. Ask for reasonable ideas, radical ideas, and ideas that just spring to mind. Once ideas are generated, group them into unusable or incomplete, valid, possible, or unlikely. Brainstorming sessions should have a leader, someone to take notes (scribe), and, of course, participants.

1.2 Functional Model

A functional model is a model of how the system is supposed to function from the user's perspective. In Agile, the functional model uses "user stories", or simple sentences (one per story) that actually describes what the user does, what they need/want, and why the need it. For example, "As a [Role], I want [feature] so that [reason/benefit]. These should be only one sentence and they are *not* for actual people. User stories do not need to be highly specialized, they just need to be enough to serve the purpose of the functional model. In effect, this means avoiding redundancy between user stories. For simple systems, all user stories should be able to fit on a single page, or two pages at absolute maximum.

After user stories are generated, move on to generating epic stories – aggregations of user stories that fit under a common theme (note: this shares some features with psychology thematic analysis).

At some point the details of how the system will be used ought to be established alongside the user stories. A good idea for such a detail is a UI mockup or, application wireframe. Often these details are put into recorded scenarios and storyboards. After these details are collected, they should be confirmed by the user – acceptance criteria can include models and user interface examples.

2 Unified Modeling Language (UML)

UML is a modeling scheme that combines use case diagrams, class diagrams, sequence diagrams, and statechart diagrams all into one unified system that ought to be understood anywhere. These diagrams are used in different parts of the process: use case diagrams are found in the functional model of requirements gathering, class diagrams are found in the object model, and sequence/interaction/activity/statechart diagrams are found in the dynamic model.

Briefly, UML is a nonprorprietary standard for modeling software systems that is a convergence of notations used in different object oriented methods: OMT, Booch, and OOSE, all developed by different people. About 80% of most problems can be diagrammed using only 20% of UML, so for brevity only use case diagrams, class diagrams, sequence diagrams, and statechart diagrams will be covered here.

All UML diagrams take the form of graphs with nodes and edges. Nodes are entities and are drawn as rectangles or ovals – rectangles denote classes or instances while ovals denote functionality. Edges between nodes represent a relationship between nodes on the graph. In use case diagrams, an actor (often represented as a stick figure (who said programmers had to be artistic?) represents a role, or a type of system user. Actors have unique, preferably short names and an optional description.

Use cases represent a class of functionality provided by the system. Use cases can be described textually, with a focus on the event flow between actors the system. The use case description should consist of 6 parts: a unique name, participating actors, entry conditions, exit conditions, flow of events, and special requirements (if any). Use cases can be related to one another (and this can likely be diagrammed in UML too), for example to represent functional behavior common to more than one use case. Child use cases can, of course, inherit behavior from parent use cases. Note that scenarios have a relationship to use cases: scenarios are instances of use cases, although not in a programming sense.

2.1 Class Diagrams

Class diagram represent the structure of a system. They are used during analysis to model application domain concepts (or objects from the user's point of view), during system design to model subsystems, and during object design to specify the details and attributes of classes from the developer's point of view.

In the code, classes should represent a concept where the class name is the only mandatory information. Each attribute has a type and each operation should have a signature. There is a certain distinction to be made between actors, classes, and objects though: actors are entities *outside* the system that are modeled whereas classes are entities inside the system that are modeled. Objects are simply instantiations of classes.

2.1.1 Notations

A list of class diagram notations follows:

- Class names and immutables have a capitalized first letter
- Abstract class names and operations are italicized. Interface names should also be italicized.
- Data type follows the variable name and a colon (e.g. isVotingAge:Bool)
- Visibility is denoted by a + for public, # for protected, - for private, and for package.

2.1.2 Relationships

As with all UML objects (?), classes can have associations between them. A line between classes can mean different things depending on the symbols at each end. Asterisks generally imply "many", so a line from one class to another with an asterisk at one end means that the class without the asterisk has a one-to-many relationship with the other, for instance. A list of relationships and symbols follows:

- Aggregation - Denotes a "consists of" hierarchy. Connections end with a rotated diamond connected to the class that maintains an aggregation of the connected objects.
- Composition - symbolized by a solid diamond. Denotes a stronger form of aggregation where the life time of the component instances is controlled by the aggregate. The parts don't exist on their own, they are entirely owned by the parent object. Aggregation means that objects are passed into a constructor, composition means that the parent object creates the components in its constructor.

- Qualifiers - Can be used to reduce the multiplicity of an association, usually implemented in either a hash table or some kind of a tree. Given as something of a branch in the relationship line that ultimately leads to the same class for both ends, but with a label on one.
- Inheritance - denotes that one class inherits another and/or builds off it. Symbolized by an empty triangle on the end of the class that is being inherited.

2.1.3 Packages

Packages help organize UML models into subsystem to increase their readability. If one were to convert a UML package into Java, you would get a literal Java package. Packages are simply represented as a series of classes and their relationships boxed into one big folder-looking thing.

2.2 Sequence Diagrams

Sequence diagrams represent behavior in terms of interactions. They are useful to identify or find missing objects, but they are very time consuming to build. Sequence diagrams mainly complement class diagrams - whereas class diagrams represent structure, sequence diagrams help show that structure in action. Sequence diagrams are primarily used during the analysis phase and can be used to refine use case descriptions. Later on, sequence diagrams can be used during the system design phase.

On paper, sequence diagrams have multiple vertical lines that represent objects with blocks along them that represent the lifetime of the object. Horizontal arrows between objects represent method calls, signals, or the creation/destruction of an object. The arrows representing interaction implicitly define methods associated with objects - the object being pointed at is the owner of the method call specified by the arrow. If an asterisk is put on one end of the arrow, it indicates iteration.

Remember not to get carried away when making sequence diagrams. Simpler diagrams are almost always preferable to complex ones. Don't make sequence diagrams more complex than they need to be; sequence diagrams are supposed to be fairly high level diagrams.

2.3 Statechart Diagrams

Statechart diagrams represent the different states that a finite state machine or a general system can go through. Shapes generally indicate a state that something can be in while arrows between them represent state transitions.

3 Analysis

In the early stages of analysis, the focus is on the interactions of users/actors with the rest of the system (focusing on such things as needed input and error handling).

3.1 Object Modeling

The main goal of object modeling is to find the most important abstractions – classes. To find them, one of the most effective methods is to go for the application domain approach, i.e. ask a domain expert to identify relevant abstractions. A second approach is a syntactic approach that looks at the use cases and analyzes the text to identify objects from the flow of events. Finally, you can just jump straight to reusable design patterns for some insight on how objects should be structured.

3.2 ECB Model

3.2.1 Entity objects

Entity objects represent persistent information tracked by the system. These are application domain objects, also called "business objects".

3.2.2 Boundary objects

Boundary objects are interfaces between actors and the system. They are typically objects such as views, forms, buttons, and hardware artifacts. Very often mockup GUIs will be modeled using boundary objects. In reality these boundary objects have no actual logic to them, they just sit there and look pretty.

3.2.3 Control objects

Control objects represent tasks performed by the system, and are mapped to a use case. These are created at the beginning of a use case and they accumulate all necessary data. Once the use case is completed, the object is destroyed. Sometimes called a "use case object".

There are different models and design patterns for working with ECB objects, namely the control-centric and the entity-centric models. In control-centric, the control object does pretty much everything: it refreshes boundary objects, it accesses an updates entites, and it recieves events from boundary objects. In entity-centric, control accesses and updates entity objects, entity publishes to boundary, and boundary pushes events to control.

The advantage to ECB is that it is possible to change one type of object without having to affect the other two types, i.e. it leads to models that are more resilient to change. These different object types actually originated in a language called Smalltalk that used an object model called MVC (model, viewm controller).

3.3 MVC Model

The MVC model is another object model that stands for "Model View Controller". Although it may seem similar to ECB, it is actually quite different. In some systems, the UI (boundary objects) is tightly coupled to data (entity objects), meaning that changes to one must result in changes to the other. In effect the UI can't be changed without changing the entity objects, and entity objects cannot be changed without changing the UI.

MVC is the solution to this problem. MVC is designed to decouple the data presentation from the data access. The data presentation subsystem is called the view, the data access subsystem is called the model, and the controller system mediates between the view and the model.

4 Object Design

The idea of object design is to prepare for implementation on the basis of design decisions. This allows certain design goals to be taken into consideration, such as performance or memory usage.

There are four major activities of object design: interface specification, reuse (of existing solutions), object model restructuring (after the 1st iteration) and object model optimization (in later iterations). During object design there is already a design in place from the analysis phase, but during object design the goal is to translate that into a more detailed object design that is written in the language of the solution. For example, a "incident report" might turn into a series of objects.

4.1 Liskov Substitution Principle

The Liskov Substitution Principle states that "If Type S can be used in all cases in place of Type T, then S is a subtype of T". This is the case with strict Inheritance, where the subclass can only add new methods to the superclass, and cannot overwrite them. This is denoted by the "final" keyword in Java.

4.2 Delegation vs. Inheritance

Code reuse can be done by delegation (where one object contains another and delegates responsibilities to it) and inheritance, but there are tradeoffs between the two. In delegation, any object can be replaced at run time by another one, but delegation is less efficient because it involves encapsulating another object. Inheritance is straightforward to use, very well supported, and it's easy to implement a new functionality, but it exposes the subclass to details of the superclass and requires recompilation anytime the parent class is changed.

4.3 Transformations

There are four different kinds of transformations when dealing with a model and code together: model transformation or remodeling, forward engineering, refactoring, and reverse engineering. Remodeling is just redoing some of the mode, forward engineering is turning that model into code, refactoring is changing the code directly, and reverse engineering is is taking those changes into account in the model.

5 Software Development Lifecycle

SDLC follows the steps of initiation, planning, execution, monitoring -. Project initiation involves creating a core team and a project charter. This project charter should involve a business justification, outline statement, economic feasibility, and technical feasibility. The initiation also involves a software size estimate and a risk assessment.

Planning involves figuring out what the organization and deliverables should be, plus the work breakdown structure, schedule (use Gantt charts), a budget, and the actual software requirements.

One way of doing a size estimate is to look to the work breakdown structure and determine size on a smaller scale from that perspective (note: experience is extremely useful here). Different quantifications exist too: lines of code (LoC), function points, feature points, number of bubbles in a data flow diagram, number of entities in ERD, or number of entities in a class diagram. Using LoC is common but has many advantages and disadvantages. LoC is very widely used and universally accepted, but it can end up applying an inappropriate pressure on developers.

During the execution phase, the software is actually required. It involves requirements elicitation, analysis (modeling), design, development, deployment, and maintenance.