Rigorous Software Engineering

Requirements Elicitation

Requirements

Requirements define features that the system must have or a constraint that it must satisfy to be accepted by the client.

The result should be a requirements specification that defines:

- Functional Requirements
  - Functionality (Relationship of outputs to inputs, exact sequence of operations, validity checks)
  - Environmental conditions (Interfaces)
  - User interaction
  - Error handling
- Nonfunctional Requirements
  - Performance
    - Static Numerical Requirements (Number of terminals supported, number of simultaneous users, ...)
    - Dynamic Numerical Requirements (Number of transactions within a certain time period)
  - Attributes (Portability, Correctness, Security ...)
  - Design Constraints
- Pseudo Requirements
  - Implementation Requirements
  - Legal Requirements

It does not define:

- System structure
- Implementation technology
- System design
- Development methodology

Requirements should be:

- Correct
- Consistent
- Complete
- Clear
- Realistic
- Verifiable
- Traceable

Requirements are usually validated using:

- Reviews
- Prototypes

Activities

Identifying Actors

- Which user groups are supported by the system?
- Which user groups execute the system's main functions?
- Which user groups perform secondary functions (maintenance, administration)?
- With what external hardware and software will the system interact?

Identifying Scenarios A narrative description of what people do and experience as they try to make use of computer systems and applications.

- What are the tasks the actor wants the system to perform?
- What information does the actor access'
- Which external changes does the actor need to inform the system about?
- Which events does the system need to inform the actor about?

Identifying Use Cases A list of steps describing the interaction between an actor and the system, to achieve a goal.

Identifying Nonfunctional Requirements Nonfunctional requirements are defined together with functional requirements because of dependencies, they typically contain conflicts.

Informal Modeling

Formal Modeling

Alloy

Modularity

Coupling

- Low coupling is a general design goal
- Each module should have a clear responsibility
- The code should still be performant and convienient
- Some design patterns increase adaptability

Data Coupling Modules get coupled by operation on shared data structures (Databases, Files).

Problems

- Changes in data structure
- Unexpected side effects (sharing by reference)
- Concurrency

Approaches

- Restricting Access to data
  - No leaking of references, clone if necessary
  - No capturing, don't store arguments as sub-object
- Making shared data immutable
  - Simplifies multi-threading
  - Easier to maintain invariants
  - Less unexpected side effects
  - Flyweight pattern
- Avoiding shared data
  - Pipe-and-filter style, no shared state, usable for streaming

Procedural Coupling Modules are coupled to other modules whose methods they call.

Problems

- Change in callee may require changes in callee

Approaches

- Moving code,
  - Duplicating functionality to avoid dependencies
- Event-based style
  - Observer pattern
    - Subject has list of observers, notifies them if an event occurs, observers decide how to handle the event
    - Model-view-controller architecture
- Restricting calls
  - Enforce a policy that restricts which other modules a module may call

Class Coupling Inheritance couples the subclass to the superclass.

Problems

- Changes in the superclass may break the subclass
- Limited options for other inheritance relations
  - Language only supports single inheritance
  - May cause conflichts with multiple inheritance

Approaches:

- Replacing inheritance with aggregation
- Using interfaces
  - Data structures can be changed without affecting the code
- Delegating allocations
  - Dependency injection
    - Dependencies between classes are defined in a separate configuration file

  - Factories
    * Delegate allocations to a dedicated class using an abstract factory
    * Different concrete factories are used to make objects of different classes
    * The concrete factory can be chosen by the client

```
// Abstract factory.
interface MapFactory<K, V> {
  Map<K, V> make();
}

// Concrete factory.
class TreeMapFactory implements MapFactory<K, V> {
  Map<K, V> make() {
    return new TreeMap<K, V>();
  }
}

class SymbolTable {
  Map<Ident, Type> types;

  // Client chooses what concrete factory is used to generate the types map.
  SymbolTyble(MapFactory<Ident, Type> f) {
    types = f.make();
  }
}
```

Adaption

- Software systems can be prepared for change by allowing clients to influence their behavior
- Designing adaptable modules makes inevitable changes easier
- Adaptable modules facilitate reuse

Parametrization Make modules parametric in:

- The values they manipulate
- The data structures they operate on
- The types they operate on
- The algorithms they apply

Specialization In object-oriented programs, behaviors can be specialized via overriding and dynamic method binding.

Overriding

```
class Merger<D> {
  Filter<D>[] filters;
  int next;
  Selector<D> s;

  D getNext() {
    D res = null;
    do {
      res = filters[next].getNext();
    } while (!s.select(res));
    next = (next + 1) % filters.length;
    return res;
  }
}
```

Drawbacks

- Subclasses share responsibility for maintaining invariants
- Increases number of possible behaviors that need to be tested
- Makes it harder to evolve code without breaking subclasses
- Performance overhead of method look-up at runtime

Dynamic Method Binding

- Can be used together with aggregation (state pattern)

```
if (s instanceof NonNullSelector) {
  s.NonNullSelector::select(res);
} else
if (s instanceof TimeStampSelector) {
  s.TimeStampSelector::select(res);
}
// Additional cases can be defined without changing the code.
```

Patterns

Functional Testing

Structural Testing

Automatic Test Case Generation

Dynamic Program Analysis

Static Program Analysis