The challenge of scaling software

- Small programs tend to be simple and malleable: relatively easy to write and to change
- Big programs tend to be complex and inflexible: harder to write and (much) harder to change
- Why? In large part because interactions become harder to understand and to manage
- We will try to reduce this challenge by using specifications to simplify and manage these interactions

A specification is a contract

- A set of obligations agreed to by the user (client) and the manufacturer (implementer) of the product
- Facilitates simplicity by two-way isolation
  - Isolate client from implementation details
  - Isolate implementer from how the part is used
  - Discourages implicit, unwritten expectations
- Facilitates change
  - Allows either side to make changes that respect the specification
  - An effective specification changes very little (at most), allowing the code (on both sides) to be more malleable

Respecting the specification has value

- If a client uses properties of the implementation that are not part of the specification, what happens if the implementation changes those properties?
- If an implementation focuses on the needs of a specific client rather than only ensuring that the specification is satisfied, what happens to other clients? To the implementation itself?

Different but dualistic roles

Implementers vs. Clients

Respecting the specification has value

- Not only in 331, but commonly in your career
- By reducing how much you and your dualistic “alter ego” know about each others’ view, the interactions can be kept cleaner
- This is hard!

You play both roles

Leading towards “Truth, Justice and the 331 Way”
Isn’t the interface a specification?

- Java (and most languages) allow programs to define interfaces as a boundary between the implementations and the clients.

```
public interface List<E> {
    public int get(int i);
    public void set(int i, E e);
    public void add(E e);
    public void add(int i, E e);
    ...
    public static boolean sub(List<T>, List<T>);
```

- The interface is a weak kind of specification that provides the syntax, but nothing about the behavior and the effects.
- This kind of contract says, “I’ll give you this and you’ll give me that, but ‘this’ and ‘that’ aren’t carefully defined.”

Why not just read code?

```
T boolean sub(List<T> src, List<T> part) {
    int part_index = 0;
    for (T elt : src) {
        if (elt.equals(part.get(part_index))) {
            part_index++;
            if (part_index == part.size()) {
                return true;
            }
        }
        else {
            part_index = 0;
        }
    }
    return false;
}
```

- In small groups, spend 1-2 minutes listing reasons why reading code would be a poor substitute for having a specification.

Code is complicated

- Much detail not needed by client — understanding every line of code is excessive and impractical.
  - Ex: Read all source code of Java libraries before using them?
- Client should care only what is in the specification, not what is in the code.
- When a client sees the implementation code, subconscious dependencies arise and may be exploited.
- Why is this bad?
- Why should you be especially concerned about this?

Why not just run code?

- The client depends on what the implementation computes — what better way to find out than by seeing what it computes?
- If you run enough test inputs, you are forming a partial specification.
  - Ex: from many standardized tests
    - “What is next in this sequence: 2, 4, 6, 8 …?”
    - “What is next in this sequence: 100, 50, 25, 76, 38, 19, 58, 29, 88, …?”
- Problems with this approach are similar to those shown in the 1st lecture via specification jeopardy.

Which code details are essential?

- A lot of choices are made in writing code — some are essential while others are incidental — but which is which?
  - Internal variable names? Algorithms used? Resource consumption (time, space, bandwidth, etc.)? Documentation? Etc.?
- Code invariably gets rewritten, making the distinction between essential and incidental crucial.
- What properties can the client rely on over time? Which properties must the implementer preserve for the client’s code to work? Future optimizations, improved algorithms, bug fixes, etc.?
- Alternatively, what properties might the implementation change that would break the client code?
- There is no simple definition of this distinction, but it is captured in practice in every specification — again, your sensibilities about this issue will grow over time.

Comments

- With more comments on comments later on.

- Comments can, and do, provide value if and when written carefully — and when kept up-to-date.
- Many comments convey only an informal, general idea of what that the code does.

```
// This method checks if “part” appears as a sub-sequence in “src”
boolean sub(List<? super T> src, List<? super T> part) {
    ...
}
```

- This usually leaves ambiguity — for example, what if src and part are both empty lists?
Improving the spec of \texttt{sub()} 

\begin{itemize}
\item Needs additional clarification
  \begin{itemize}
  \item a) src and part cannot be null
  \item b) If src is empty list, always returns false
  \item c) Results may be unexpected if partial matches can happen
  \item d) right before a real match: e.g., list \{1,2,1,3\} will not \texttt{be identified as a sub sequence of} \{1,2,1,3,3\}
  \end{itemize}
\item Or needs to be replaced with a more detailed description
  \begin{itemize}
  \item This method scans the \texttt{src} list from beginning to end.
  \item building up a match for \texttt{"part"}, and resetting that match every time that...
  \end{itemize}
\end{itemize}

Further improving the spec of \texttt{sub()} 

\begin{itemize}
\item A complicated description suggests poor design and rarely clarifies a specification
\item Try to simplify rather than describe complexity
\item Perlis: Simplicity does not precede complexity, but follows it."
\item Rewrite the specification of \texttt{sub()} more clearly and sensibly
  \begin{itemize}
  \item \texttt{// returns true iff sequences A, B exist such that \texttt{src = A : part : B \texttt{";\texttt{ 是 sequence concatenation}}}}
  \item The “declarative” style of this specification is important
  \item Contrast to an operational style such as “This method scans the \texttt{src} list from beginning to end.”
  \item The mathematical flavor is not necessary, but it can help reduce ambiguity
\end{itemize}
\end{itemize}

Examples of specifications 

\begin{itemize}
\item Javadoc “is a tool for generating API documentation in HTML format from doc comments in source code.”
\item Get used to using it
\item Javadoc conventions expect programs to provide
  \begin{itemize}
  \item method prototype – basically, the name of the method and the types of the parameters and of the return
  \item text description of method
  \item @param: description of what gets passed in
  \item @returns: description of what gets returned
  \item @throws: list of exceptions that may occur
  \end{itemize}
\end{itemize}

Example: Javadoc for String.contains

\begin{itemize}
\item \texttt{tags} in Java comments
\item These are parsed and formatted by Javadoc
\item Viewable in web browsers
\end{itemize}

CSE 331 specifications
(Javadoc is extensible) 

\begin{itemize}
\item The \texttt{precondition}: constraints that hold before the method is called
  \begin{itemize}
  \item \texttt{requires}: spells out any obligations on client (if \texttt{requires} is not satisfied by a client, the implementation is unconstrained)
  \end{itemize}
\item The \texttt{postcondition}: constraints that hold after the method is called (if the \texttt{precondition} held)
  \begin{itemize}
  \item \texttt{modifies}: lists objects that may be affected by method; any object not listed is guaranteed to be untouched
  \item \texttt{throws}: lists possible exceptions
  \item \texttt{effects}: gives guarantees on the final state of modified objects
  \item \texttt{returns}: describes return value
  \end{itemize}
\end{itemize}

Ex 1: Spec and an implementation

\begin{itemize}
\item \texttt{stat}ic \texttt{int test(List\texttt{T}\texttt{;} list, T oldelt, T newelt)} \texttt{requires \texttt{list}, \texttt{oldelt}, and \texttt{newelt} are non-null oldelt occurs in list}
  \texttt{modifies \texttt{list}}
  \texttt{effects: change the first occurrence of \texttt{oldelt} in \texttt{list} to \texttt{newelt}}
  \texttt{no other changes to \texttt{list}}
  \texttt{returns: position of element in \texttt{list} that was \texttt{oldelt} and is now \texttt{newelt}}
\end{itemize}
Ex 2: Spec and an implementation

```java
static List<Integer> listAdd(List<Integer> lst1, List<Integer> lst2) {
    List<Integer> res = new ArrayList<>();
    for (int i = 0; i < lst1.size(); i++) {
        res.add(lst1.get(i) + lst2.get(i));
    }
    return res;
}
```

Ex 3: Spec and an implementation

```java
static void listAdd2(List<Integer> lst1, List<Integer> lst2) {
    for (int i = 0; i < lst1.size(); i++) {
        lst1.set(i, lst1.get(i) + lst2.get(i));
    }
}
```

Ex 4: Spec and an implementation

```java
static void uniquify(List<Integer> lst) {
    for (int i=0; i < lst.size() - 1; i++)
        if (lst.get(i) == lst.get(i+1))
            lst.remove(i);
}
```

In small groups, spend 1-2 minutes filling in the ?? in the specification above

Improved specification

```java
public static int binarySearch(int[] a, int key) {
    requires: a is sorted in ascending order
    returns: some i such that a[i] = key if such an i exists, otherwise -1
    ```

Summary

- Properties of a specification
  - The client relies only on the specification and on nothing (else) from the implementation
  - The implementer provides everything in the specification and is otherwise unconstrained
- Overall, effective use of specifications leads to simpler and more flexible programs that have fewer bugs and cleaner dependencies
Next steps

- Assignment 0
  - Due today 11:59PM
- Assignment 1
  - Out later today
  - Due Wednesday (10/5) 11:59PM
- Assignment 2
  - Out Wednesday (10/5)
    - Due in two parts
    - Part A on Friday (10/7) 11:59PM
    - Part B the following Wednesday (10/12) at 11:59PM
- Lectures
  - Testing and Junit (M)
  - Equality (W)
  - Abstract data types I (F)
  - Abstract data types II (M)