ECE 321 Software Requirements Engineering

Lecture 14: Design by Contract

Design by Contract

- Programming methodology that aims to
  - Write specification/design early
  - Smoothly transfer the specification/design to implementation
- Is based on pre and postconditions
  - The idea is to fail fast if a precondition is not met

Underlying idea

- Every software component satisfies a certain goal, for the benefit of other components
- This goal can be seen as part of the component’s contract
  - It ‘promises’ to deliver a particular thing

Example: Contract for snail mail delivery

Contract between:
Client
Post office

Example: Contract for snail mail delivery

Contract between:	**Obligations**	**Benefits**
Client		
Post office		

Example: Contract for snail mail delivery

Contract between:	**Obligations**	**Benefits**
Client	(Satisfy precondition) Bring package before 4pm, and pay delivery fee	
Post office	(Satisfy postcondition) Deliver package by 10am next day	

Example: Contract for snail mail delivery

Contract between:	**Obligations**	**Benefits**
Client	(Satisfy precondition) Bring package before 4pm, and pay delivery fee	(From postcondition) Package is delivered by 10am next day
Post office	(Satisfy postcondition) Deliver package by 10am next day	(From precondition) Not required to do anything if package brought after 4pm or fee not paid

Design by contract in software

- Software components also have obligations and benefits
  - This becomes clear when you view them as supplier and client
- The definitions of the obligations and benefits form the **contracts** between components

The contract

- Binds two (or more) parties
  - i.e., supplier and client
- Is explicit (= written)
- Specifies all mutual obligations and benefits
  - No hidden clauses

The contract in a software system

- Represented by preconditions, postconditions and invariants in a class
- The parties are methods, and clients that call the methods

  Method x

  - Requires preconditions
  - Ensures postconditions
  - If you promise to call x with the preconditions satisfied, then x promises to deliver a final state in which the postconditions are satisfied

Example: Contract for pop for a stack

Example: Contract for pop for a stack

Contract between:
Client
Pop

Example: Contract for pop for a stack

Contract between:	Obligations	Benefits
Client		
Pop		

Example: Contract for pop for a stack

Contract between:	**Obligations**	**Benefits**
Client	Call pop() only for a non-empty stack	
Pop	Make sure that the top element is removed from the stack, and the number of elements on the stack is decreased by 1	

Example: Contract for pop for a stack

Contract between:	**Obligations**	**Benefits**
Client	Call pop() only for a non-empty stack	The top of the stack is removed and returned, and the number of elements on the stack is decreased by 1
Pop	Make sure that the top element is removed from the stack, and the number of elements on the stack is decreased by 1	Do not bother with cases where the stack is empty

Example: Contract for pop for a stack

Contract between:	**Obligations**	**Benefits**
Client	Call pop() only for a non-empty stack	The top of the stack is removed and returned, and the number of elements on the stack is decreased by 1
Pop	Make sure that the top element is removed from the stack, and the number of elements on the stack is decreased by 1	Do not bother with cases where the stack is empty

Note that contracts are also helpful for specifying where responsibilities are in a system

Contract elements

- Preconditions
- Postconditions
- Invariants
- For all elements holds
  - They only CHECK values, never modify
  - They can be implemented by one or more assert or if-statements

Contract elements: Preconditions

- Check if input arguments are valid
- The check can be anything
- Uses input variables, state of variables, etc.
- Example:
  - Stack is non-empty before pop() is called

Contract elements: Postconditions

- Check if result is valid before it is passed back
- Uses returned values, input arguments, state of variables before/after executing function, etc.
- Example:
  - Size of stack is decreased by 1 after pop() is called

Contract elements: Invariants

- Checks if the variables of a method always have valid values
  - Usually in Design by Contract we talk about class invariants
- Often implemented inside a method that is executed before and after each method call
- Example:
  - Size of the stack is never negative

How do we write a contract?

- Define the preconditions and postconditions for each method
  - At a small scale they are usually logical
  - They can be checked during testing
  - Sometimes we can mathematically prove them
- Invariants exist to help the user get a mental image of how a class works
  - So use them to your advantage

How do we enforce contracts?

- Develop the contracts while working on specification
- ‘Real’ Design by Contract approaches design the contract first
- Embed the contract in the code

What happens if a precondition is not satisfied?

- If client’s part of the contract is not fulfilled, a method can do what it pleases:
  - Return any value
  - Loop indefinitely
  - Terminate in some wild way
- Note: postcondition is usually precondition for another method... so has the same effect

Which languages support Design by Contract?

- The Eiffel language has built-in support
  - Not really a popular language these days
- Other languages have more limited support
  - Java, C/C++ use assert statements + specialized compilers
  - Several libraries exist that help with the implementation

Example: iContract library in Java

```
/**
 * @pre f >= 0.0
 * @post Math.abs((return * return) - f) < 0.001
 */
public float sqrt(float f) { ... }
```

Though this is fairly easy to implement
with assertions yourself...

Benefits of Design by Contract

- Leads to precise and correct specifications that can easily be transferred to implementation
  - Development process becomes more focused
- Provides interface documentation that is always up to date
- Results in faults that occur close to their cause

Course wrap-up

- This was the final ‘official’ lecture of the course
- There are two more ‘unofficial’ lectures left

What did we discuss? 1/3

- Software life cycle
  - Waterfall, evolutionary, spiral models
- Software quality
  - External, internal
- Software requirements: introduction
  - Problem versus solution domain
  - Proper requirements
  - Sources of information

What did we discuss? 2/3

- The five steps of problem analysis
- Software requirements: elicitation
  - Interviewing, workshops, use cases, prototyping
- Vision document
- SRS document
- Specification styles
  - Declarative, operational

What did we discuss? 3/3

- UML
- Algebraic specifications
- Finite state machines
- Petri Nets
- Design by Contract

Final exam

- Friday December 14, 9am-11am
- ETLC E2 001

About the next two lectures

- I propose to do the following:
  - Nov 29: extra office hours during lecture time
  - Dec 6: Discuss old exam
    - I will post it before Nov 29
    - You could use the free time on Nov 29 to make the exam :) and ask questions on Dec 6