CS189A Capstone
Fall 2023

Lecture 6: Software Design & Modularization

PRD: Product Requirements Document

- The official statement of what is required of the system developers
- Includes a specification of both user and system requirements

- Defines WHAT the system should do, not HOW it should do it
  - Design comes later

- Agile and extreme SWE processes express requirements as
  - Use cases – how a system will act
  - Or as scenarios called user stories (describe result/benefit of it)

  - **Both** document how the system responds from an *external* perspective (when viewed from the **outside**) – like a black box…

So - we are only interested in describing externally visible behavior

PRDv1: Your **Living** Requirements Document: A Shared Google Doc (due tomorrow)

- Authors, Team, Project Title
- Intro – including problem, innovation, science, core technical advance (2-3 pages)
  - Define project specifics, team goals/objectives, background, and assumptions
- System architecture overview
  - High level diagram (1 page)
  - User interaction and design (1 page)
- Requirements (functional and non-functional)
  - User stories or use cases (links): 10 for PRDv1 prioritized
  - Prototyping code, tests, metrics (5+ user stories): github commits/issues
- System models: contexts, sequences, behavioral/UML, state
- Appendices
  - Technologies employed

On-going Process

- Evolving (aka "living") requirements document
  - Identify/learn (and teach each other) the technologies required
  - Write user stories in particular; update the requirements as you go:
    - Prioritize stories and mark **mandatory, important, or desirable**
    - Assign time estimates to stories; improve your estimation ability over time
    - Specify **acceptance test** for each story – should be in code
- Concurrently as part of Sprint
  - Break down stories into tasks (begin design/prototyping process)
    - Prioritize tasks
    - Assign timings to tasks
    - Specify what (code) test(s) are to be used as evidence of task completion/acceptance
    - Each member/developer chooses task, implements, and tests task
    - Another member does code review/test and accepts the pull request
      - Test is the one specified above (Acceptance)
    - When a Story is complete, some member performs story test/acceptance

PRDv2: due next week

- Authors, Team, Project Title
- Intro: problem, innovation, science, core technical advance
  - Define project specifics, team goals/objectives, background, and assumptions
- System architecture overview
  - High level diagram (1 page)
  - User interaction and design (1 page)
- Requirements (functional and non-functional)
  - User stories or use cases (links): 20+ for PRDv2 ***prioritized***
  - Prototyping code, tests, metrics (10+ user stories): github commits/issues
- System models (1+ pages)
  - Contexts, interactions, structural, behavioral (UML)
  - Use cases, sequencing, event response, system state, classes/objects
- Appendices - Technologies employed

Software Design

- We can think of software design in two main phases
  - Architectural Design
    - Divide the system into a set of modules
    - Decide the interfaces of the modules
    - Figure out the interactions among different modules
  - Detailed Design
    - Detailed design for individual modules
    - Write the pre and post-conditions for the operations in each module
    - Write pseudo code for individual modules explaining key functionality

Fundamental Principles

There are some fundamental principles in software engineering:

- **Anticipation of Change**
  - We talked about this a lot in the context of software process models. The main principle behind agile software development.
- **Separation of Concerns**
  - You can see the use of this principle in the requirements analysis and specification. For example: separating functional requirements from performance requirements.
- **Modularity**
  - This is what I will talk about today as it applies to software design
- **Iterative (Stepwise) Refinement**
  - For example separating architectural design from detailed design
- **Abstraction**
  - We will see examples of this when we discuss design patterns

Modularity

- Modularity principle suggests dividing a complex system into simpler pieces, called modules
- When we have a set of modules we can use separation of concerns and work on each module separately
- Modularity can also help us to create an abstraction of a modules environment using interfaces of other modules

Modularization

- According to Parnas
  - *"... modularization as a mechanism for improving the flexibility and comprehensibility of a system while allowing the shortening of its development time."*

- The goals of modularization are to
  - reduce the complexity of the software
  - and to improve
    - maintainability
    - reusability
    - productivity

Benefits of Modularization

- Managerial (productivity)
  - development time should be shortened because several groups work on different modules with limited need for communication
- Product flexibility (reusability, maintainability)
  - it should be possible to make changes to one module without the need to change others
- Comprehensibility (reducing complexity)
  - it should be possible to study the system one module at a time

Modularization

- Gouthier and Pont:

  "*A well-defined segmentation of the project effort ensures system modularity. Each task forms a separate, distinct program module. At implementation time each module and its inputs and outputs are well-defined, there is no confusion in the intended interface with other system modules. At checkout time the integrity of the module is tested independently ... Finally, the system is maintained in modular fashion; system errors and deficiencies can be traced to specific system modules, thus limiting the scope of detailed error searching*."

Modularization

- A module is a responsibility assignment rather than a subprogram
- Question: What are the criteria to be used in dividing the system into modules?

Modularization

- In dividing a system into modules we need some guiding principles.
  - What is good for a module?
  - What is bad for a module?

- There are two notions which characterize good things and bad things about modules nicely
  - Cohesion
    - We want highly cohesive modules
  - Coupling
    - We want low coupling between modules

Cohesion and Coupling

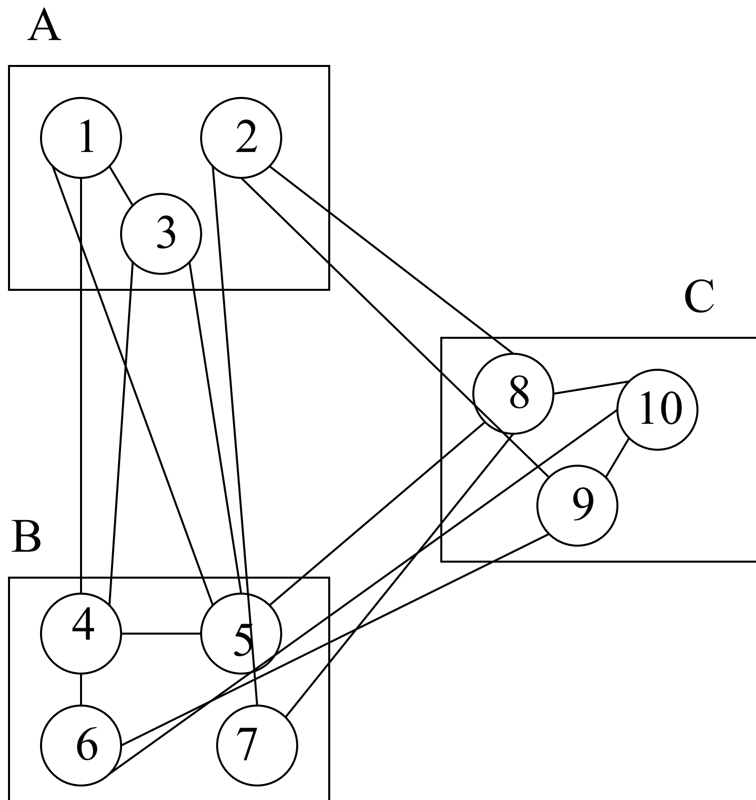
- What is cohesion?
  - Type of association among different components of a module
  - Cohesion assesses why the components are grouped together in a module

- What is Coupling?
  - A measure of strength of interconnection (the communication bandwidth, the dependencies) between modules
  - Coupling assesses the kind and the quantity of interconnections among modules

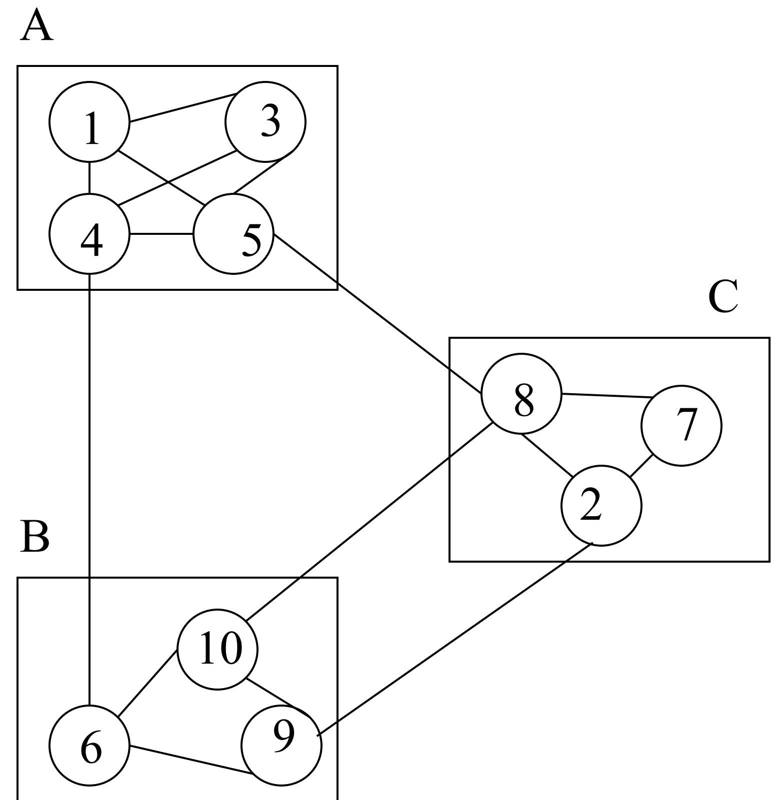
- Good modularization:
  - high cohesion and low coupling

- Modules with high cohesion and low coupling lead to less bugs and are easier to fix

Cohesion and Coupling



Bad modularization:
low cohesion, high coupling



Good modularization:
high cohesion, low coupling

Types of Cohesion

- There are various informal categorizations of cohesion types in a module. I will discuss some of them (starting with the ones which are considered low cohesion)

- WORST: Coincidental cohesion
  - Different components are thrown into a module without any justification, i.e., they have no relation to each other
    - Maybe this was the last module where all the remaining components were put together
  - Obviously, this type of cohesion is not good! It basically corresponds to lack of cohesion.

Types of Cohesion

- BAD: Logical cohesion
  - A module performs multiple somewhat related operations one of which is selected by a control flag that is passed to the module
  - It is called logical cohesion because the control flow (i.e. the “logic”) of the module is the only thing that ties the operations in the module together

```
procedure operations (data1, data2, operation)
{
  switch (operation) {
    case ...: // execute operation 1 on data1
    case ...: // execute operation 2 on data2
  }
}
```

Types of Cohesion

- BAD: Temporal cohesion
  - A module performs a set of functions related in time
    - For example an initialization module performs operations that are only related by time
  - These operations can be working on different data types
  - A user of a module with temporal cohesion can not call different operations separately

```
procedure initialize_game()
{
    // initialize the game board
    // set players' scores to 0
}
```

Types of Cohesion

- Coincidental, logical and temporal cohesion should be avoided.
- Such modules are hard to debug and modify.
- Their interfaces are difficult to understand.

Types of Cohesion

- BETTER: Communicational cohesion
  - Grouping a sequence of operations that operate on the same data in the same module
  - Some drawbacks: Users of the module may want to use a subset of the operations.

```
procedure operations1and2 (data)
{
    // execute operation 1 on data
    // execute operation 2 on data
}
```

Types of Cohesion

- GOOD: Functional cohesion
  - Every component within the module contributes to performing a single function
  - Before object orientation this was the recommended approach to modularization.
  - No encapsulation between a data type and operations on that data type

```
procedure operation1 (data)
{
    // execute operation 1 on data
}

procedure operation2 (data)
    // execute operation 2 on data
}
```

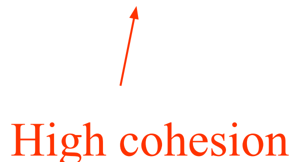
Types of Cohesion

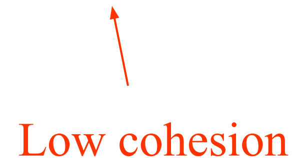
- BEST: Informational Cohesion
  - This term is made up to mean the data and functionality encapsulation used in object oriented design

```
module stack
// definition of the stack data type
procedure initialize() { .. }
procedure pop() { .. }
procedure push() { .. }
procedure top_element() { .. }
```

- A ranking of (from good to bad) types of cohesion:

informational > functional > communicational > temporal > logical > coincidental





Types of Coupling

- Coupling is the type and amount of interaction between modules
- Coupling among modules
  - module A and B access to the same global variable
  - module A calls module B with some arguments

- Arbitrary modularization will result with tight coupling
  - Loosely coupled modules are good, tightly coupled modules are bad

- If you use only one module you get no coupling. Is this a good idea?
  - No! You did not reduce the complexity of the system. You did not modularize.

Types of Bad Coupling

- Common (or Global) coupling
  - Access to a common global data by multiple modules
  - Class variables are also a limited form of common coupling, use them with caution
- This is a bad type of coupling: The interactions among the modules are through global data so it is very difficult to understand their interfaces and interactions. It is hard to debug, and maintain such code.

```
int number_of_students
procedure find_maximum_grade(student_grades)
{
// traverse the array student_grades from 0 to number_of_students
// to find the maximum grade
}
procedure find_minimum_grade(student_grades)
{
// traverse the array student_grades from 0 to number_of_students
// to find the minimum grade
}
```

Types of Bad Coupling

- Control coupling
  - If one module passes an element of control to another module
  - For example a flag passed by one module to another controls the logic of the other module
- This type of code is hard to understand
  - It is hard to understand the interfaces among the modules, you need to look at the functionality to understand the interfaces

```
call operations (d1, d2, opcode);

procedure operations (data1, data2, operation)
{
  switch (operation) {
    case ...: // execute operation 1 on data1
    case ...: // execute operation 2 on data2
  }
}
```

Good coupling

- Data coupling
  - The interaction between the modules is through arguments passed between modules
  - The arguments passed are homogenous data items

- Data coupling is the best type of coupling

- In the data coupling you should try to pass only the parts of data that is going to be used by the receiving module
  - do not pass redundant parts

Modularization

- Complexity
  - A design with complex modules is worse than a design with simpler modules
  - Remember the initial motivation in modularization is to reduce the complexity
  - If your modules are complex this means that you did not modularize enough
  - Modularization means using divide-and-conquer approach to reduce complexity

Modularization

- Now we will discuss and compare two modularization strategies
- These modularization strategies are both intended to generate modules with high cohesion and low coupling
  - Modularization technique 1: Functional decomposition
  - Modularization technique 2: Parnas's modularization technique

"On the Criteria to be Used in Decomposing Systems into Modules", Parnas 1972

Modularization: Functional Decomposition

- Functional decomposition
  - Divide and conquer approach
  - Use stepwise refinement
    1. Clearly state the intended function
    2. Divide the function to a set of subfunctions and re-express the intended function as an equivalent structure of properly connected subfunctions, each solving part of the problem
    3. Divide each subfunction far enough until the complexity of each subfunction is manageable
- How do you divide a function to a set of subfunctions? What is the criteria? This approach does not specify the criteria for decomposition.
  - Based on how you decompose the system the modules will show different types of cohesion and coupling

Modularization: Functional Decomposition

- One way of achieving functional decomposition: Make each step in the computation a separate module
  - Draw a flowchart showing the steps of the computation and convert steps of the computation to modules
  - Shortcoming: Does not specify the granularity of each step

- Another way of achieving functional decomposition is to look at the data flows in the system
  - Represent the system as a set of processes that modify data. Each process takes some data as input and produces some data as output.
  - Each process becomes a module

- Shortcoming: Both of these approaches produce a network of modules, not a hierarchy

What about Data Structures?

- Fred Brooks: *“Show me your code and conceal your data structures, and I shall continue to be mystified. Show me your data structures, and I won’t usually need your code; it’ll be obvious.”*
- Eric Stevens Raymond:*“Smart data structures and dumb code works a lot better than the other way around.”*

- Functional decomposition focuses on the operations performed on data
- According to Brooks and Raymond data structures should come first

Parnas's Criteria Modularization

- In a famous paper Dave Parnas proposed a modularization principle that became one of the core ideas in object oriented design:

  "On the Criteria To Be Used in Decomposing Systems into Modules" by D. L. Parnas., Communications of the ACM, Volume 15, Issue 12, December 1972, pp. 1053-1058.

Parnas's Criteria Modularization: Information Hiding

- Every module in the decomposition is characterized by its knowledge of a design decision which it hides from others.
  - Its interface or definition is chosen to reveal as little as possible about its inner workings
  - This principle is called **Information Hiding**

- Modules do not correspond to steps in the computation

- A data structure, its internal representation, and accessing and modifying procedures for that data structure are part of a single module

Modularization a la Parnas = Object Oriented Design

- In his paper on modularization, which pre-dates object-oriented programming languages, Parnas advocates principles of object-oriented design and programming
  - Information hiding
  - Encapsulation: encapsulate data and the functionality
  - Abstraction: One module can be a specialization of another module
  - Inheritance: One module can inherit functionality from another module
  - Polymorphism: An instance of a module or an instance of its specialization can be both used by some other module without rewriting separate code
- All of these features are supported by modern object-oriented languages such as C++ and Java

What about Efficiency?

- There will be too many procedure calls in the second approach which may degrade the performance
  - Use inline expansion, insert the code for a procedure at the site of the procedure call to save the procedure call overhead
  - This is a common compiler optimization

Modularization Summary

- The goals of modularization are to ***reduce the complexity*** of the software, and to ***improve maintainability, reusability and productivity***.
- A module is a responsibility assignment rather than a subprogram.
- Good modularization: ***highly cohesive modules*** and ***low coupling between modules***
- One modularization approach:
  - ***Functional decomposition***: Draw a flowchart showing the steps of the computation and convert steps of the computation to modules.
- Better modularization approach:
  - ***Information hiding***: Isolate the changeable parts, make each changeable part a secret for a module. Module interface should not reveal module's secrets.

Designing for Change

In another famous paper, Parnas advocates designing for change:

“Designing Software for Ease of Extension and Contraction. IEEE Trans. Software Eng. 5(2): 128-138 (1979)”

This expands on the information hiding principle and advocates for writing programs as a family of programs

Designing for Change

Common complaints about software systems:

- It is not possible to deliver an early release with a subset of the intended capabilities since the subset does not work until everything works
- Adding a new capability requires rewriting most of the code
- It is not possible to simplify the system and improve its performance by removing unwanted capability since to take advantage of the simplification major parts of the system have to be rewritten
- We are not able customize the software system based on the client's needs

Software as a Family of Programs

- When you are designing a program you should think that you are designing a family of programs
- The ways the members of the program family could differ
  - They may run on different hardware configurations
  - They may perform the same functions but differ in the format of the input and output data
  - They may differ in certain data structures or algorithms because of differences in available resources
  - They may differ in data structures or algorithms because of differences in the size of the data sets or the relative frequency of of certain events
  - Some users may require only a subset of the services or features that other users need, and they may not want to be forced to pay for the resources consumed by the unneeded features

Why Are Some Programs Not Subsetable or Extensible?

- Excessive information distribution
  - Too many parts of the system knows about a particular design decision.
  - If that design decision changes all the parts that “know” about it have to be changed
- A chain of data transforming components
  - If the system is designed as a chain of components each receiving data from the previous component, processing it and changing the input format before sending the data to the next program in the chain.
  - If one component in the chain is not needed, it is hard to remove since the output of its predecessor is not compatible with its successor

Why Are Some Programs Not Subsetable or Extensible?

- Components that perform more than one function
  - Combining more than one function in the same component.
  - The functions can be simple and they may be combined without difficulty, however, extension and contraction of the resulting program will be difficult

Designing for Change

- Requirements definition
  - Identify possible subsets of components in the system that can perform a useful function
  - Search for the minimal subset that can provide a useful service and then search for a set of minimal increments which provide additional functionality

Designing for Change

- Information hiding
  - Isolate the changeable parts in modules
  - Develop an interface between modules and the rest of the system that remains valid for all versions
  - Crucial steps
    - Identify the items that are likely to change, these will be the "secrets"
    - Design intermodule interfaces that are insensitive to the anticipated changes. The changeable aspects (i.e., the "secrets") should not be revealed by the interface

Designing for Change

- The virtual machine concept:
  - Bad habits
    - Do not think of components as steps in processing, this goal oriented thinking can be natural but it is bad if you are designing for change
    - Do not think that you are writing programs that perform transformations from input data to output data
  - Think that you are creating a virtual machine by extending the capabilities given to you by the hardware or the programming language
    - You are extending the data types with additional data types and you are extending the instructions by providing instructions that operate on the data types you defined

Design by Contract

- Design by Contract and the language that implements the Design by Contract principles (called Eiffel) was developed in Santa Barbara by Bertrand Meyer (he was a UCSB professor at the time)
- Bertrand Meyer won the 2006 ACM Software System Award for the Eiffel!
  - Award citation: "*For designing and developing the Eiffel programming language, method and environment, embodying the Design by Contract approach to software development and other features that facilitate the construction of reliable, extendible and efficient software*."
- The company which supports the Eiffel language is located in Santa Barbara:
  - Eiffel Software (http://www.eiffel.com)
- The material in the following slides is mostly from the following paper:
  - "Applying Design by Contract," B. Meyer, IEEE Computer, pp. 40-51, October 1992.

Dependability and Object-Orientation

- An important aspect of object oriented design is reuse
  - For reusable components correctness is crucial since an error in a module can affect every other module that uses it

- Main goal of object oriented design and programming is to improve the quality of software
  - The most important quality of software is its dependability

- Design by contract presents a set of principles to produce dependable and robust object oriented software
  - Basic design by contract principles can be used in any object oriented programming language

What is a Contract?

- There are two parties:
  - Client which requests a service
  - Supplier which supplies the service

- Contract is the agreement between the client and the supplier

- Two major characteristics of a contract
  - Each party expects some benefits from the contract and is prepared to incur some obligations to obtain them
  - These benefits and obligations are documented in a contract document

- Benefit of the client is the obligation of the supplier, and vice versa.

What is a Contract?

- As an example let's think about the contract between a tenant and a landlord

Party	Obligations	Benefits
Tenant	Pay the rent at the beginning of the month.	Stay at the apartment.
Landlord	Keep the apartment in a habitable state.	Get the rent payment every month.

What is a Contract?

- A contract document between a client and a supplier protects both sides
  - It protects the client by specifying how much should be done to get the benefit. The client is entitled to receive a certain result.
  - It protects the supplier by specifying how little is acceptable. The supplier must not be liable for failing to carry out tasks outside of the specified scope.

- If a party fulfills its obligations it is entitled to its benefits
  - No Hidden Clauses Rule: no requirement other than the obligations written in the contract can be imposed on a party to obtain the benefits

How Do Contracts Relate to Software Design?

- You are not in law school, so what are we talking about?

- Here is the basic idea
  - One can think of pre and post conditions of a procedure as obligations and benefits of a contract between the client (the caller) and the supplier (the called procedure)

- Design by contract promotes using pre and post-conditions (written as assertions) as a part of module design

- Eiffel is an object oriented programming language that supports design by contract
  - In Eiffel the pre and post-conditions are written using require and ensure constructs, respectively

Design by Contract in Eiffel

In Eiffel procedures are written is in the following form:

```
procedure_name(argument declarations) is
    -- Header comment
require
    Precondition
do
    Procedure body
ensure
    Postcondition
end
```

Design by Contract in Eiffel

An example:

```
put_child(new_child: NODE) is
    -- Add new to the children of current node
require
    new_child /= Void
do
    ... Insertion algorithm ...
ensure
    new_child.parent = Current;
    child_count = old child_count + 1
end  -- put_child
```

- `Current` refers to the current instance of the object (`this` in Java)
- `Old` keyword is used to denote the value of a variable on entry to the procedure
- Note that "=" is the equality operator (== in Java) and "/=" is the inequality operator (! = in Java)

The put_child Contract

- The put_child contract in English would be something like the table below.
  - Eiffel language enables the software developer to write this contract formally using require and ensure constructs

Party	Obligations	Benefits
Client	Use as argument a reference, say `new_child`, to an existing object of type `Node`.	Get an updated tree where the `Current` node has one more child than before; `new_child` now has `Current` as its parent.
Supplier	Insert `new_child` as required.	No need check if the argument actually points to an object.

Contracts

- The pre and postconditions are assertions, i.e., they are expressions which evaluate to true or false
  - The precondition expresses the requirements that any call must satisfy
  - The postcondition expresses the properties that are ensured at the end of the procedure execution

- If there is no precondition or postcondition, then the precondition or postcondition is assumed to be true (which is equivalent to saying there is no pre or postcondition)

Assertion Violations

- What happens if a precondition or a postcondition fails (i.e., evaluates to false)
  - The assertions can be checked (i.e., monitored) dynamically at run-time to debug the software
  - A ***precondition violation*** would indicate a bug at the ***caller***
  - A ***postcondition violation*** would indicate a bug at the ***callee***

- Our goal is to prevent assertion violations from happening
  - The pre and postconditions are not supposed to fail if the software is correct
    - hence, they differ from exceptions and exception handling
  - By writing the contracts explicitly, we are trying to avoid contract violations, (i.e, failed pre and postconditions)

Assertion Violations

- In the example below, if `new_child = Void` then the precondition fails.
- The procedure body is not supposed to handle the case where `new_child = Void`, that is the responsibility of the caller

```
put_child(new_child: NODE) is
    -- Add new to the children of current node
require
    new_child /= Void
do
    ... Insertion algorithm ...
ensure
    new_child.parent = Current;
    child_count = old child_count + 1
end  -- put_child
```

Defensive Programming vs. Design by Contract

- Defensive programming is an approach that promotes putting checks in every module to detect unexpected situations

- This results in redundant checks (for example, both caller and callee may check the same condition)
  - A lot of checks makes the software more complex and harder to maintain

- In Design by Contract the responsibility assignment is clear and it is part of the module interface
  - prevents redundant checks
  - easier to maintain
  - provides a (partial) specification of functionality

Class Invariants

- A class invariant is an assertion that holds for all instances (objects) of the class
  - A class invariant must be satisfied after creation of every instance of the class
  - The invariant must be preserved by every method of the class, i.e., if we assume that the invariant holds at the method entry it should hold at the method exit
  - We can think of the class invariant as conjunction added to the precondition and postcondition of each method in the class

- For example, a class invariant for a binary tree could be (in Eiffel notation)

```
invariant
  left /= Void implies (left.parent = Current)
  right /=Void implies (right.parent = Current)
```

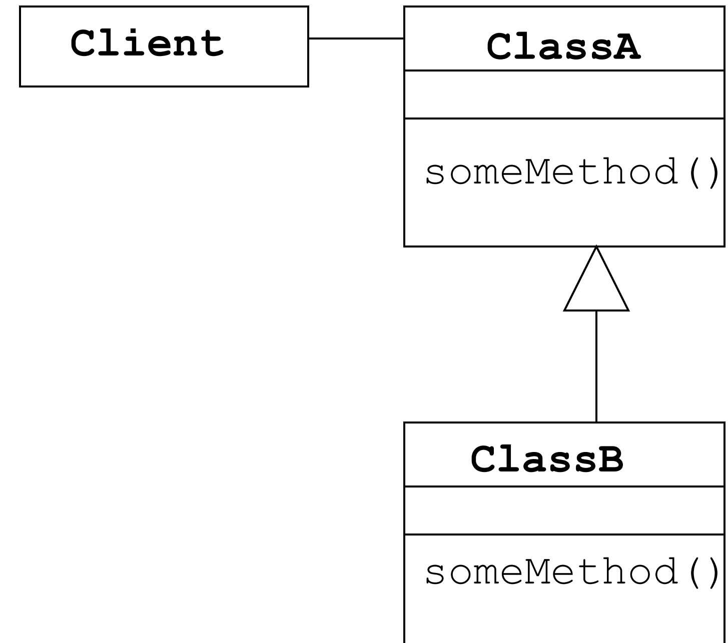
Design by Contract and Inheritance

- Inheritance enables declaration of subclasses which can redeclare some of the methods of the parent class, or provide an implementation for the abstract methods of the parent class
- Polymorphism and dynamic binding combined with inheritance are powerful programming tools provided by object oriented languages
  - How can the Design by Contract can be extended to handle these concepts?

Inheritance: Preconditions

- If the precondition of the `ClassB.someMethod` is stronger than the precondition of the `ClassA.someMethod`, then this is not fair to the `Client`

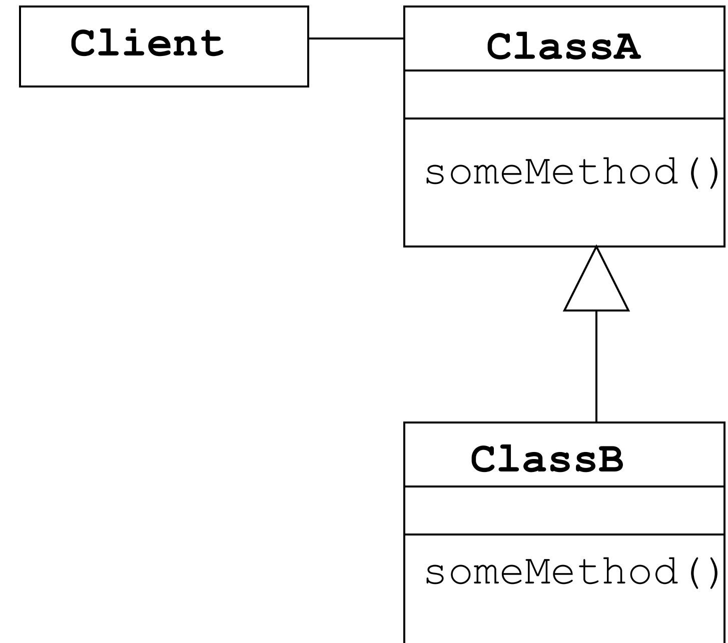
- The code for `ClassB` may have been written after `Client` was written, so `Client` has no way of knowing its contractual requirements for `ClassB`



Inheritance: Postconditions

- If the postcondition of the `ClassB.someMethod` is weaker than the postcondition of the `ClassA.someMethod`, then this is not fair to the `Client`

- Since `Client` may not have known about `ClassB`, it could have relied on the stronger guarantees provided by the `ClassA.someMethod`

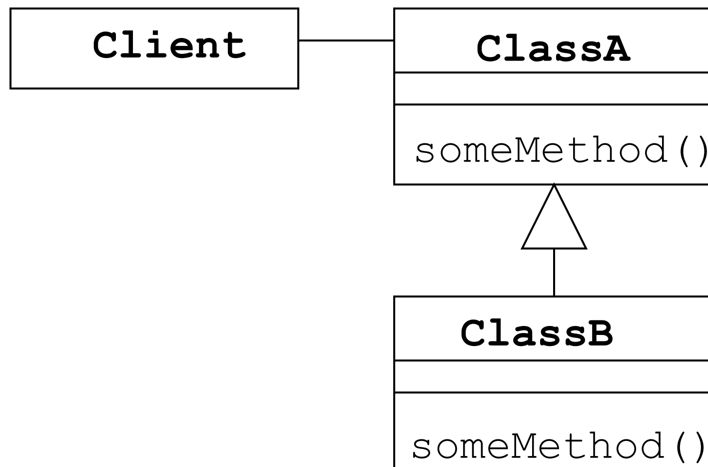


Inheritance

- Eiffel enforces the following
  - the precondition of a derived method to be weaker
  - the postcondition of a derived method to be stronger
- In Eiffel when a method overwrites another method the new declared precondition is combined with previous precondition using disjunction
- When a method overwrites another method the new declared postcondition is combined with previous postcondition using conjunction
- Also, the invariants of the parent class are passed to the derived classes
  - invariants are combined using conjunction

In `ClassA`:

```
invariant
    classInvariant
someMethod() is
require
    Precondition
do
    Procedure body
ensure
    Postcondition
end
```



In `ClassB` which is derived from `ClassA`:

```
invariant
    newClassInvariant
someMethod() is
require
    newPrecondition
do
    Procedure body
ensure
    newPostcondition
end
```

The precondition of `ClassB.aMethod` is defined as:
`newPrecondition` **or** `Precondition`

The postcondition of `ClassB.aMethod` is defined as:
`newPostcondition` **and** `Postcondition`

The invariant of `ClassB` is
`classInvariant` **and** `newClassInvariant`

Dynamic Design-by-Contract Monitoring

- Enforce contracts at run-time
- A contract
  - Preconditions of modules
    - What conditions the module requests from the clients
  - Postconditions of modules
    - What guarantees the module gives to clients
  - Invariants of the objects
- Precondition violation, the client is to blame
  - Generate an error message blaming the client (caller)
- Postcondition violation, the server is to blame
  - Generate an error message blaming the server (callee)

Design by Contract

- Java Modeling Language (JML) is an annotation language for Java that enables specification of contracts for Java classes as annotations. There contract checking tools built based on JML
- Design by contract C++ is an implementation of Eiffel's design by contract approach for C++
- There have been tools implemented for design by contract checking (statically) and monitoring (dynamically)
  - ESC/Java: Static verification of pre, post-condition and class invariant violations
  - jContractor (developed at UCSB): Dynamic contract monitoring for detecting pre, post-condition and invariant violations at runtime

Writing Contacts in UML+OCL

- Object Constraint Language (OCL) is a specification language that supports specification of contracts (i.e., pre, post conditions and invariants) in UML class diagrams.

- OCL constraints have formal syntax and semantics
  - their interpretation is unambiguous

- OCL can be used to add precision to UML diagrams

- There are tools which check OCL constraints.:
  - USE (A UML-based Specification Environment)
    https://sourceforge.net/projects/useocl/
  - Enables analysis of UML diagrams before implementation

Writing Contracts in UML + OCL