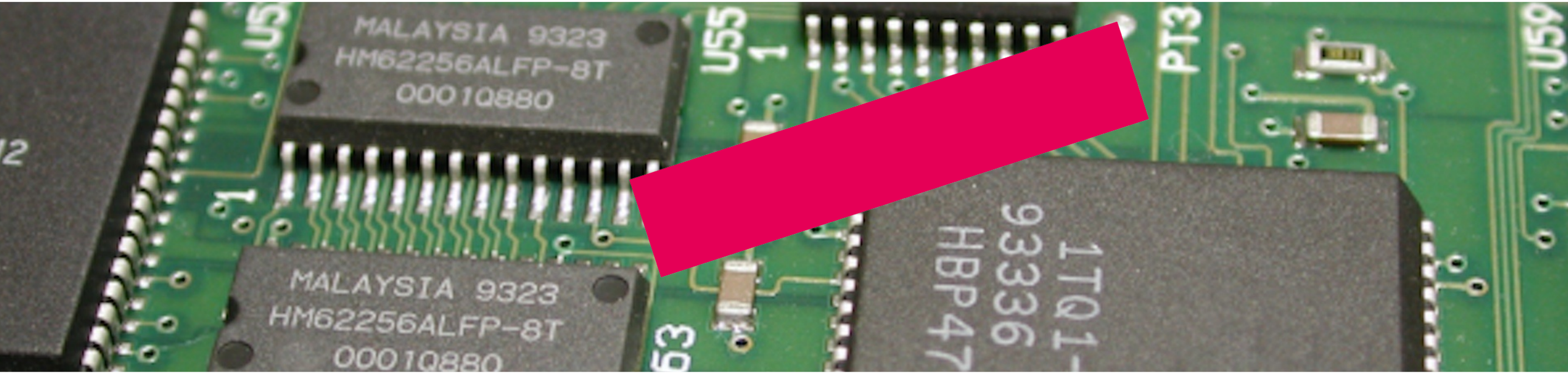
Embedded Systems Development - 3. Lists, inline functions, default params.



Electrical Engineering / Embedded Systems
Faculty of Engineering

Johan.Korten@han.nl

HAN_UNIVERSITY
OF APPLIED SCIENCES

To begin with…

Attendance

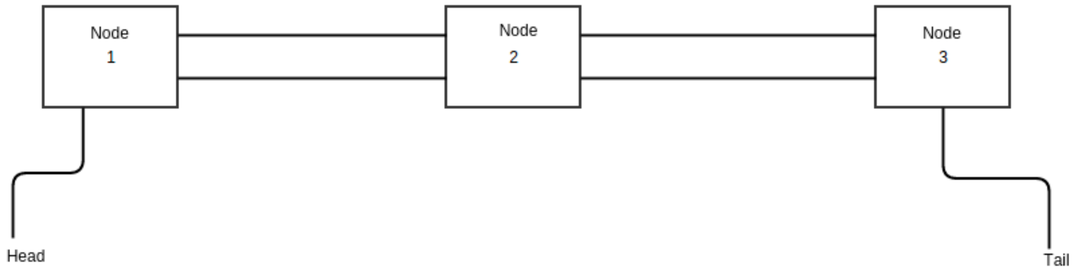
Schedule (exact info see #OO and roster at insite.han.nl)

		C++	UML
Step 1	Single responsibility	Scope, namespaces, string	Class diagram
Step 2	Open-Closed Principle	Constructors, iterators, lambdas	Inheritance / Generalization
Step 3	**Liskov Substitution Principle**	**lists, inline functions, default params**	**Activities**
Step 4	Interface Segregation Principle	interfaces and abstract classes	Sequence diagrams
Step 5	Dependency Inversion Principle	threads, callback	
Step 6	Coupling and cohesion	polymorphism	
Step 7	n/a	n/a	n/a

Note: subject to changes as we go…

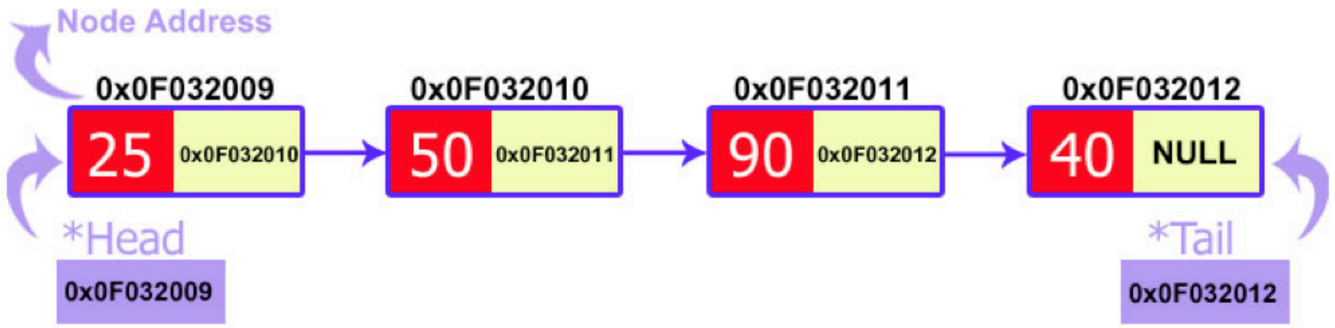
C++ Language: Lists vs. Vectors

	Vector	List
Memory	contiguous	noncontiguous
Space allocation	Pre-allocation for future	No pre-allocation for future
Thread safety	Yes	No
Speed of addition / deletion	Take more CPU	Are faster
Searching	Very fast	Much slower
Overall safety	Very safe	Not so much
Large elements	Works ok	Good support



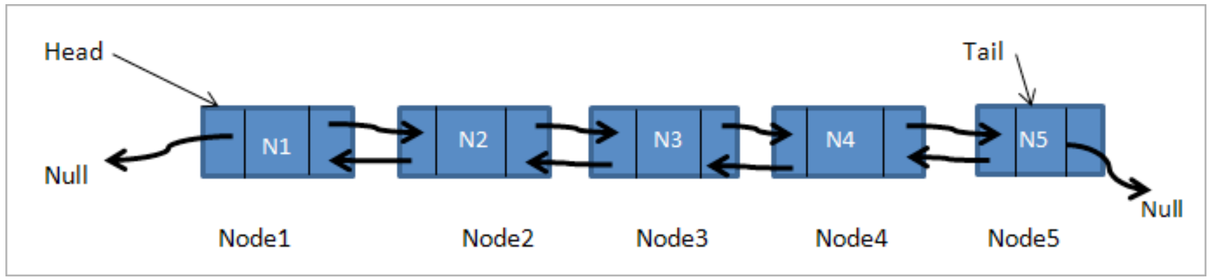
C++ Language: Singly Linked Lists

https://www.codementor.io/@codementorteam/a-comprehensive-guide-to-implementation-of-singly-linked-list-using-c_plus_plus-ondlm5azr



C++ Language: Doubly Linked Lists

https://www.softwaretestinghelp.com/doubly-linked-list/



C++ Language: Linked List Example

```cpp
#include <iostream>
using namespace std;

struct Node {
   int data;
   struct Node *next;
};

struct Node* head = NULL;
void insert(int new_data) {
   struct Node* new_node = (struct Node*) malloc(sizeof(struct Node));
   new_node->data = new_data;
   new_node->next = head;
   head = new_node;
}

void display() {
   struct Node* ptr;
   ptr = head;
   while (ptr != NULL) {
      cout<< ptr->data <<" ";
      ptr = ptr->next;
   }
}
```

```cpp
int main() {
   insert(3);
   insert(1);
   insert(7);
   insert(2);
   insert(9);
   cout << "The linked list is: ";
   display();
   cout << endl;
   return 0;
}


// output:

The linked list is: 9 2 7 1 3
```

https://www.tutorialspoint.com/cplusplus-program-to-implement-singly-linked-list

C++ Language: Default parameters

```
#include<iostream>
using namespace std;


// A function with default arguments, it can be called with
// 2 arguments or 3 arguments or 4 arguments.
int sum(int x, int y, int z=0, int w=0)
{
    return (x + y + z + w);
}

/* Driver program to test above function*/
int main()
{
    cout << sum(10, 15) << endl;
    cout << sum(10, 15, 25) << endl;
    cout << sum(10, 15, 25, 30) << endl;
    return 0;
}
```

C++ Language: Inline functions

Replacement for macros (C)

***inline**:*

- compiler will inserting the function code at the address of each function call
- linker behavior is changed as well

- pro: ‘inlining’ saves overhead of a function call
- con: duplicate code in program (larger binaries)

C++ Language: Inline functions Example

```
// declaration of inline function
static inline void swap(int *m, int *n)
{
    int tmp = *m;
    *m = *n;
    *n = tmp;
}

// call of inline function
swap(&x, &y);

// compiler might replace call by:
int tmp = x;
x = y;
y = tmp;
```

SOLID: Liskov Substitution Principle

On subtypes:

“What is wanted here is something like the following substitution property:

If for each object o1 of type S there is an object o2 of type T
such that for all programs P defined in terms of T,

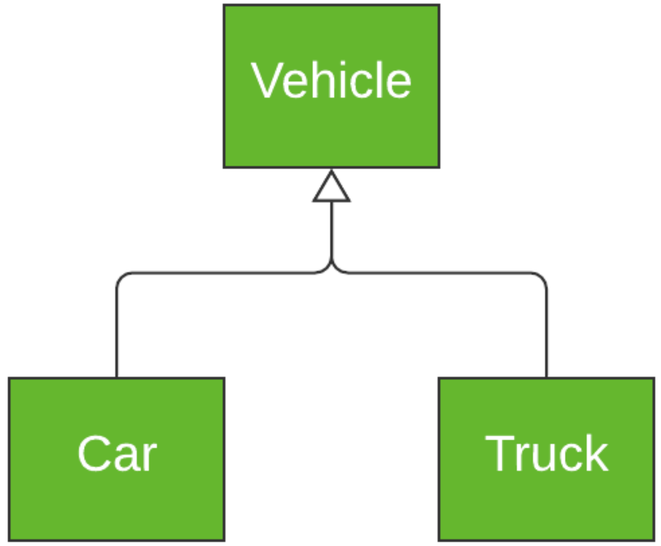
the behavior of P is unchanged when o1
is substituted for o2

then S is a subtype of T.”

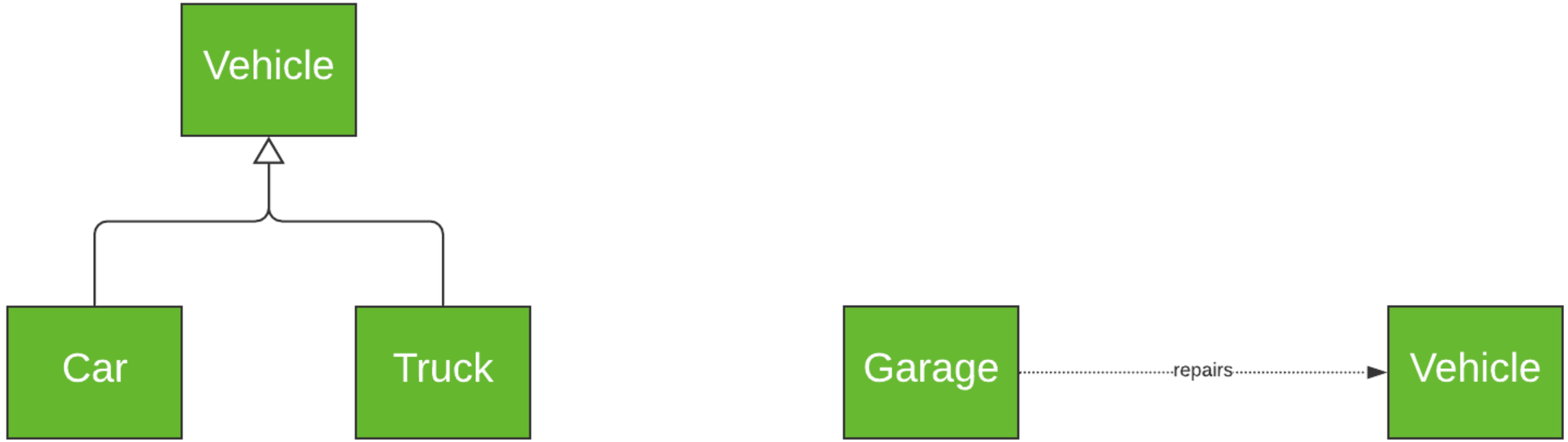
- Meyer 1988

(From: Clean Architecture, R.C. Martin, 2018)

SOLID: Liskov Substitution Principle



SOLID: Liskov Substitution Principle



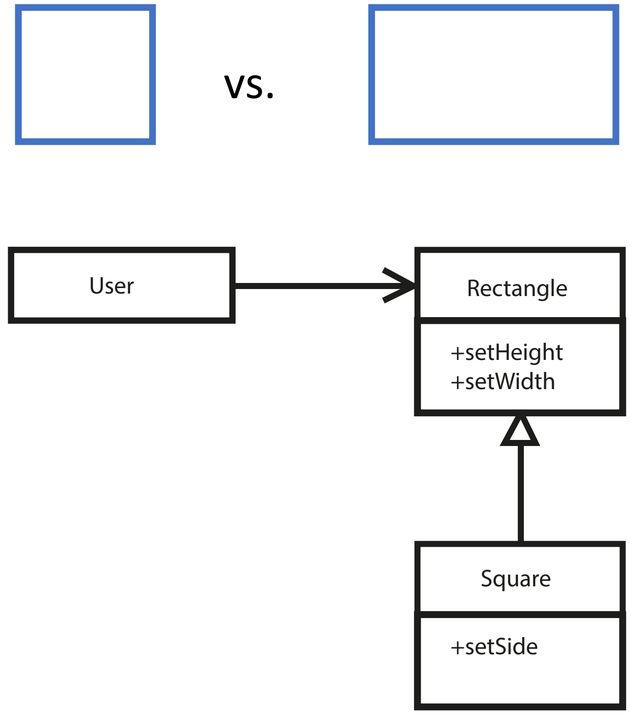
Liskov means: Garage repairs vehicles, both cars and trucks…

SOLID: Liskov Substitution Principle - Squares and Rectangles

```
#include <math.h>

// Derived classes
class Rectangle: public Shape {
   public:
      void setHeight(int _height) {
         height = _height;
      }
      void setWidth(int _width) {
         width = _width;
      }
};

class Square: public Rectangle {
   public:
      void setSide(int _side) {
         setHeight(_side);
         setWidth(_side);
      }
};
```

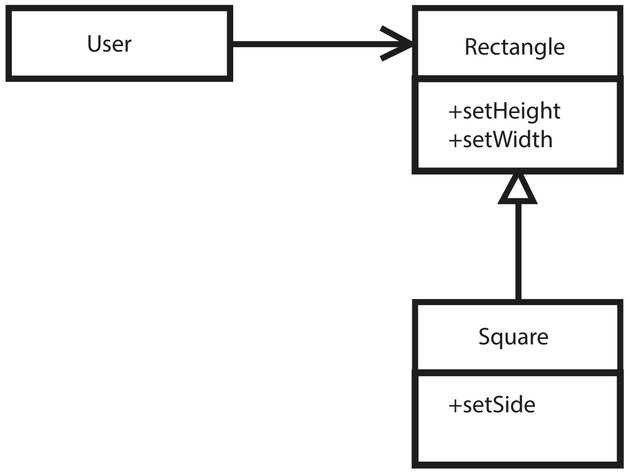


SOLID: Liskov Substitution Principle

User thinks it is communicating with a rectangle so would assume the following would be valid:

```
Rectangle r = ...
r.setWidth(5);
r.setHeigth(2);
assert(r.getArea() == 10)
```

Where could we take measures to prevent this from happening?



SOLID: Liskov Substitution Principle

```
class Shape {
   public:
      virtual int getArea() = 0;
      void setWidth(int w) {
         width = w;
      }

      void setHeight(int h) {
         height = h;
      }

   protected:
      int width;
      int height;
};
```

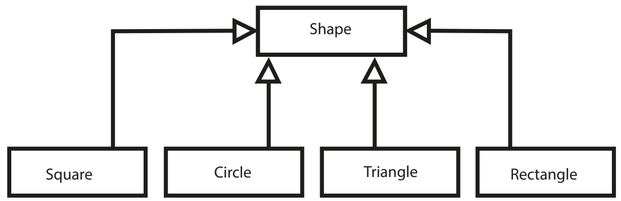
So was this in fact a good idea?!

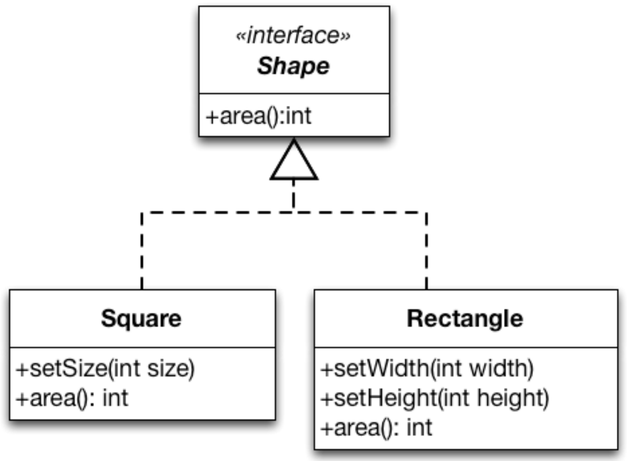
SOLID: Liskov Substitution Principle

```
class Shape {
   public:
      virtual int getArea() = 0;

  protected:
      int width;
      int height;
};
```

It would be better to design it like this:





SOLID: Liskov Substitution Principle: summary

So in fact it is not enough that instances of SomeSubclassA and SomeSubclassB provide all methods declared in SomeClass but ***these methods should also behave like their heirs****.*

A client method ***should not be able to distinguish the behavior of objects*** *of SomeSubclassA and SomeSubclassB from that of objects of SomeClass.*

As such the Liskov Substitution Principle (LSP) is an extension of the Open/Closed principle.

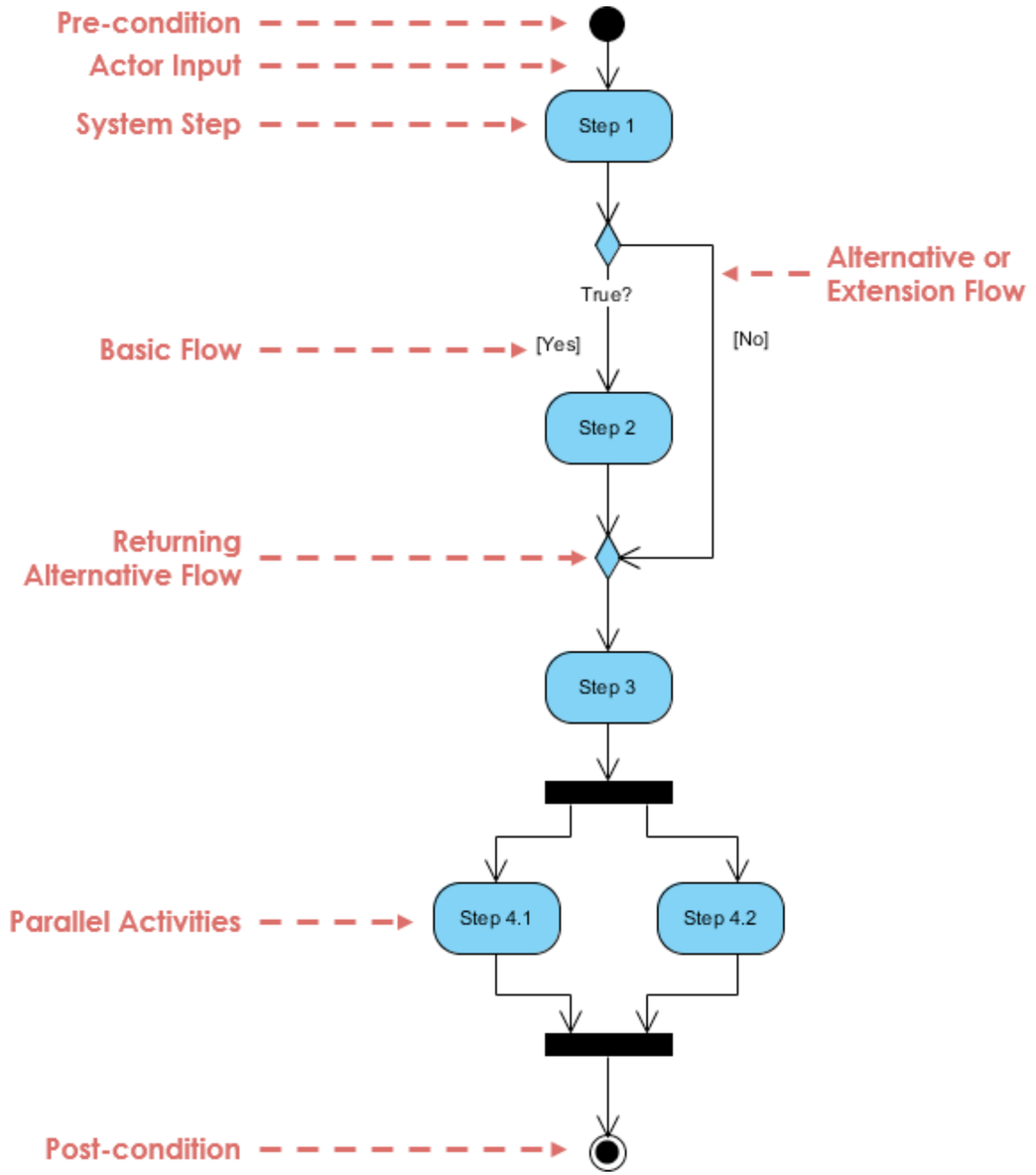
http://stg-tud.github.io/sedc/Lecture/ws13-14/3.3-LSP.html#mode=document

UML: Activities

Closely related to state diagrams

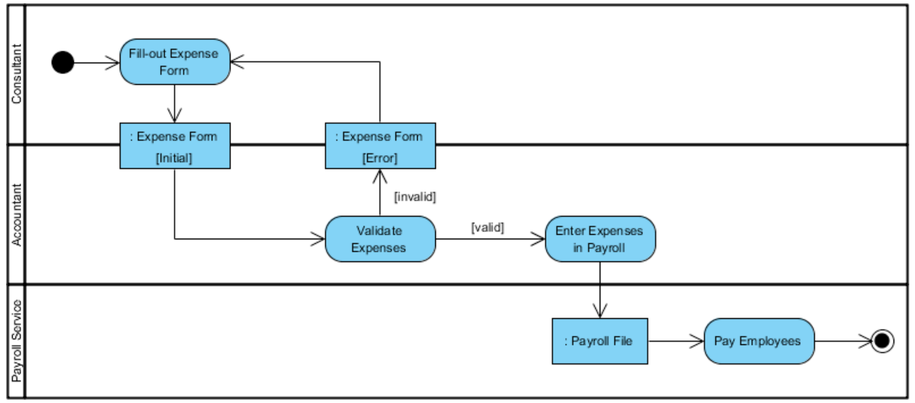
States: mainly internal actions (that might have external representation)

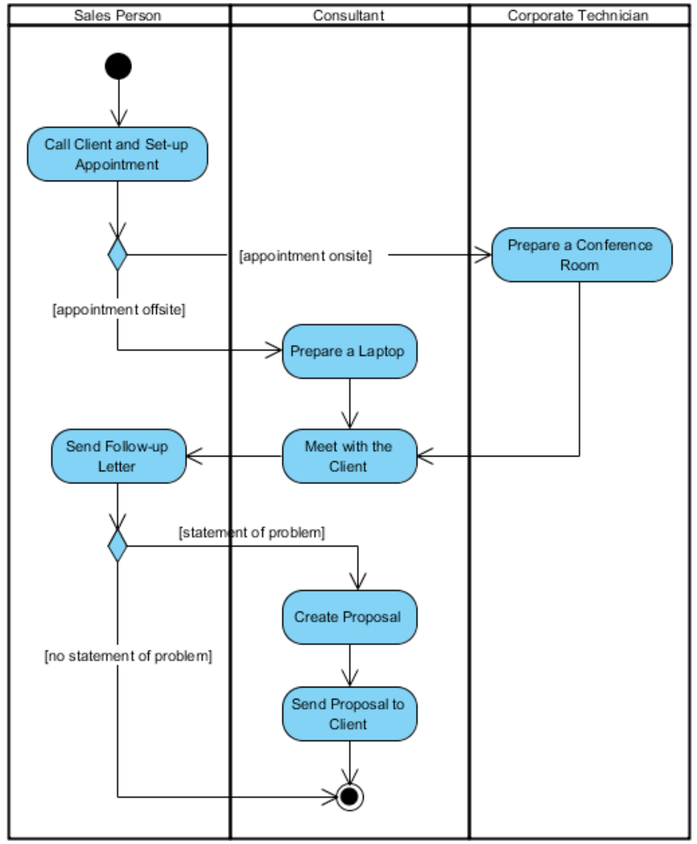
Activities: mostly external (user) actions, e.g. for HMI



UML: Activities in Swimlanes

Horizontal or Vertical
Lanes: actors





On Design Patterns

Taken from: Bruce Douglas, 2011, Design Patterns for Embedded Systems in C

- **Accessing hardware:**
  - Hardware proxy pattern
  - Hardware adapter pattern
  - Mediator pattern
  - Observer pattern
  - Debouncing pattern
  - Interrupt pattern
  - Polling pattern

- Embedded Concurrency and Resource management
- State machine patterns
- Safety and reliability patterns

Hardware Proxy pattern

Hardware Proxy Pattern creates software element responsible for access to a piece of hardware and encapsulation of hardware compression and coding implementation (Douglas, 2011, p.85).

The simple implementation of this pattern doesn’t implement anything for thread safety. It can be combined with the Critical Region, Guarded Call, or Queuing patterns to provide thread safety. For deadlock avoidance, it can be combined with the Ordered Locking or Simultaneous Locking patterns.

Hardware Adapter pattern

The Hardware Adapter Pattern provides a way of adapting an existing hardware interface into the expectations of the application.

The Hardware Adapter extends the Hardware Proxy Pattern. The latter pattern encapsulates hardware interface detail but does not translate service requests into radically different ones. The Hardware Adapter Pattern adds a level of indirection between the client and the Hardware Proxy. This allows the clients to be unchanged in the application while at the same time permitting reuse of existing Hardware Proxies that may also have been created for other systems. The implementation of the Hardware Proxy and the Hardware Adapter can be merged, but that undermines the reusability of the Hardware Proxy.

Mediator pattern

The Mediator Pattern provides a means of coordinating a complex interaction among a set of elements.

The Mediator Pattern is particularly useful for managing different hardware elements when their behavior must be coordinated in well-defined but complex ways. It is particularly useful for C applications because it doesn't require a lot of specialization (subclassing), which can introduce its own complexities into the implementation.

Many embedded applications control sets of actuators that must work in concert to achieve the desired effect. For example, to achieve a coordinated movement of a multi-joint robot arm, all the motors must work together to provide the desired arm movement. Similarly, using reaction wheels or thrusters in a spacecraft in three dimensions requires many different such devices acting at precisely the right time and with the right amount of force to achieve attitude stabilization.

This pattern creates a mediator that coordinates the set of collaborating actuators but without requiring direct coupling of those devices.

This pattern is similar to the Strategy Pattern architectural Rendezvous Pattern (Douglas, 2003). In this case we are focusing on its use for detailed hardware coordination.

Observer pattern

The Observer Pattern is one of the most common patterns around. When present, it provides a means for objects to “listen in” on others while requiring no modifications whatsoever to the data servers. In the embedded domain, this means that sensor data can be easily shared to elements that may not even exist when the sensor proxies are written.

The observer pattern relies on ‘radio stations’, keeping track of the listening ‘stations’ could be implemented using some list-like structure, in C++ preferably a vector.

This pattern can be freely mixed with the previous patterns in this chapter since its concerns are orthogonal. In embedded systems it is very common to add Observer functionality to a Hardware Proxy or Hardware Adapter, for example.

Debouncing Pattern

This simple pattern is used to reject multiple false events arising from intermittent contact of metal surfaces.

The pattern is often used in conjunction with the Interrupt Pattern discussed elsewhere in this chapter. Timeouts are often shown within a tm() event on the state machine.

Interrupt Pattern

The physical world is fundamentally both concurrent and asynchronous; it's nonlinear too, but that's a different story. Things happen when they happen and if your embedded system isn’t paying attention, those occurrences may be lost. Interrupt handlers (a.k.a. Interrupt Service Routines, or ISRs) are a useful way to be notified when an event of interest occurs even if your embedded system is off doing other processing.

An alternative means to get hardware signals and data is to periodically check for them via the Polling Pattern.

Polling Pattern

The Polling Pattern addresses the concern of getting new sensor data or hardware signals into the system as it runs when the data or events are not highly urgent and the time between data sampling can be guaranteed to be fast enough.

Periodic polling is a special case of the Interrupt Pattern. In addition, the hardware checks may be done by invoking data acquisition services of Hardware Proxies or Hardware Adapters. In addition, the Observer Pattern can be merged in as well, with the polling element (OpportunisticPoller or PeriodicPoller) serving as the data server and the PollDataClients serving as the data clients.

Links

https://www.visual-paradigm.com/guide/uml-unified-modeling-language/what-is-activity-diagram/

https://github.com/ksvbka/design_pattern_for_embedded_system/blob/master/design-patterns-for-embedded-systems-in-c-an-embedded-software-engineering-toolkit.pdf

https://www.baeldung.com/cs/liskov-substitution-principle

Any questions?