ST2 cheatsheet

- Observer -

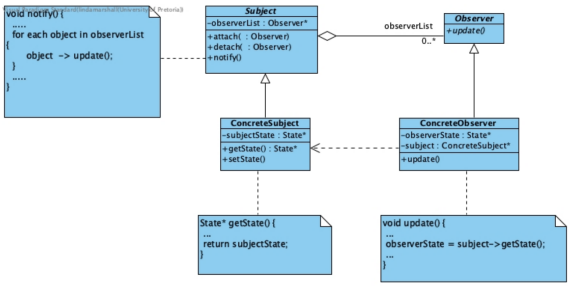
The Observer Design Pattern is a behavioral design pattern used in software engineering. It is primarily used to establish a one-to-many dependency between objects, where one object (called the subject or the publisher) maintains a list of its dependents (observers or subscribers) and notifies them of any state changes, typically by calling one of their methods.

1. **Purpose**: The Observer pattern is used to create a mechanism where multiple objects (observers) are notified when the state of another object (the subject) changes. This helps in achieving loose coupling between the subject and observers.
2. **Participants**:
   - **Subject**: This is the object being observed. It maintains a list of observers and provides methods to add, remove, and notify observers.
   - **Observer**: These are the objects that want to be notified of changes in the subject. They typically implement an interface or inherit from a base class that defines the update method.
3. **How it works**:
   - When the subject's state changes, it iterates through its list of observers and calls their update method.
   - Observers can then react to the change in the subject's state as needed.

4. **Benefits**:
    - **Flexibility**: It allows you to add or remove observers without changing the subject's code.
    - **Decoupling**: The subject doesn't need to know specific details about its observers, promoting loose coupling.
    - **Broadcasting**: Changes in the subject can be broadcasted to multiple observers.
5. **Example**: Think of a weather station as the subject. It collects weather data and has multiple displays (observers) like a temperature display, humidity display, and so on. When the weather data changes, all displays are automatically updated with the new information.
6. **Common Usage**: Observer pattern is widely used in GUI frameworks (like Swing in Java), event handling systems, and in implementing the publish-subscribe mechanism.
7. **Drawbacks**: One potential drawback is that if an observer doesn't unsubscribe properly, it can cause memory leaks. Also, overuse of this pattern can lead to complexity.

- **Classification**: Behavioural design pattern
- **Intent:** Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically

Structure



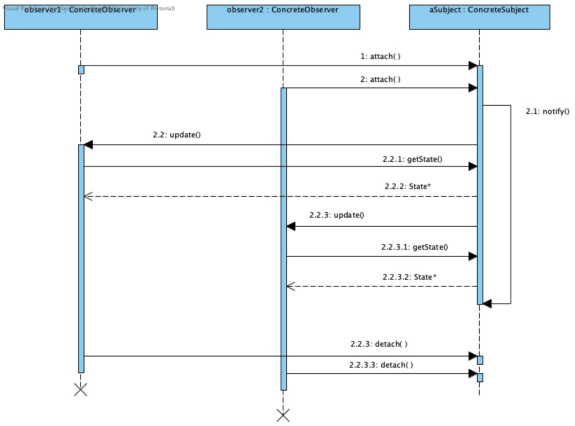
UML diagram of Observer pattern

Hierarchies

- **Subject hierarchy**: objects that are to be observed
- **Observer hierarchy**: objects that do observation

Participants

- **Subject**: Provides interface for observers to



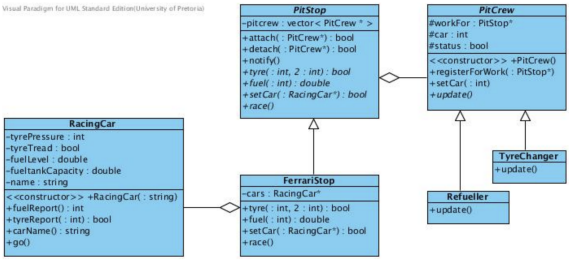
Sequence diagram

Related patterns

- **Mediator**: Promotes loose coupling, enables indepent object interaction and state transfer
- **Singleton**: Ensures subject has only one access point to itself

Examples

- **Consider Grand Prix:**
  - Pit crew for changing tires and refuelling
    - Car - **subject**
    - Pit crew - **observer**



attach and detach to concrete subject

- **ConcreteSubject**:
  - Implementation of subject being observed
  - Implements functionality to store objects that are observing it and sends update notifications to these objects
- **Observer**:
  - Defines interface of objects that may observe subject
  - Provides means by which observer are notified regarding change to subject
- **ConcreteObserver**:
  - Mantains reference to subject it observes
  - Updates and stores relevant state info of subject in order to keep consistent with state of the subject

-Participants enables attaching and detaching of observers from the code and leaving subject intact.

-Observers of subject register with the subject and will be notified when a change occurs in the subject

```
bool PitStop::attach(PitCrew  person) {
  pitcrew.push back(person);
  person− > registerForWork(this);
  return true;
};

bool PitStop::detach(PitCrew  person) {
  bool found = false;
  vector < PitCrew  > ::iterator it = pitcrew.begin();
  while ((it != pitcrew.end()) && (!found)) {
    if ( it == person) {
      found = true;
      pitcrew.erase(it);
    }
    ++it;
  }
  return found;
}
void PitStop::notify() {
  vector < PitCrew  > ::iterator it = pitcrew.begin();
  for (it = pitcrew.begin(); it != pitcrew.end(); ++it) {
    ( it)− > update();
  }
}
bool FerrariStop::tyre(int car, int tyre) {
  return cars[car− 1]− > tyreReport(tyre− 1);
}
double FerrariStop::fuel(int car) {
  return cars[car− 1]− > fuelReport();
}
bool FerrariStop::setCar(RacingCar  car) {
  static int carId = 0;
  if (carId < 2) {
```

```
    cars[carId] = car;
    carId++;
    return true;
  }
  return false;
}
void FerrariStop::race() {
  int input;
  cout << ”Type in a number[0 stops]”;
  cin >> input;
  while (input != 0) {
    if ((input % 2) == 0) {
      cars[0]− > go();
    } else {
      cars[1]− > go();
    }
    printWorkshopStatus(this);
    notify();
    cout << ”Type in a number[0 stops]”;
    cin >> input;
  }
}
```

```
void PitCrew::registerForWork(PitStop  employer) {
  workFor = employer;
}
void PitCrew::setCar(int c) {
  car = c;
}
void TyreChanger::update() {
  if (status == 0) {
    cout << ”Check tyre status” << endl;
    bool tyreStatus = false;
    for (int i = 1; i <= 4; i++)
      tyreStatus = tyreStatus && workFor− > tyre(car, i);
    if (tyreStatus) {
      status = 1;
      cout << ”Need to change all tyres” << endl;
    }
  } else
    status = 0;
}
void Refueller::update() {
  cout << ”Refeuller
  for car” << car << ”status is” << status << endl;
  if (status == 0) {
    cout << ”Check fuel status” << endl;
    double fuelStatus = workFor− > fuel(car);
    cout << ”fuel status is: ” << fuelStatus << endl;
    if (fuelStatus < 20) {
      status = 1;
      cout << ”Need to add fuel” << endl;
    }
```

```
  } else
    status = 0;
}
```

```
RacingCar::RacingCar(string n): name(n) {
  for (int i = 0; i < 4; i++) {
    tyrePressure[i] = 4;
    tyreTread[i] = true;
  }
  fueltankCapacity = 100;
  fuelLevel = 100;
}
int RacingCar::fuelReport() {
  return fuelLevel / fueltankCapacity 100;
}
bool RacingCar::tyreReport(int tyre) {
  return tyrePressure[tyre] && tyreTread[tyre];
}
string RacingCar::carName() {
  return name;
}
void RacingCar::go() {
  int input;
  cout << ”Type in any value: ” << endl;
  cin >> input;
  if ((input % 2) == 0) {
    // Do the tyres
    if ((input % 3) == 0) {
      tyreTread[input % 4] = false;
    } else {
      tyrePressure[input % 4] = false;
    }
  } else {
    // Do the fuel
    fuelLevel− = 5;
  }
}
void printWorkshopStatus(PitStop p) {
  cout << ”Fuel
  for car 1 = ” << p− > fuel(1) << endl;
  cout << ”Fuel
  for car 2 = ” << p− > fuel(2) << endl;
  for (int i = 1; i <= 4; i++) {
    cout << ”Tyre
    for car 1, tyre” << i << ” = ” << p− > tyre(1, i) << endl;
    cout << ”Tyre
    for car 2, tyre” << i << ” = ” << p− > tyre(2, i) << endl;
  }
}
```

- Iterator -

The Iterator Design Pattern is a behavioral design pattern used to provide a way to access the elements of an aggregate (collection) object sequentially without exposing its underlying representation.

1. **Purpose**: The Iterator pattern helps you to traverse through elements of a collection (like a list or an array) without needing to know the internal structure of that collection.
2. **Participants**:
   - **Iterator**: This is an interface or a class that defines methods like `next()`, `hasNext()`, and possibly `remove()`. It keeps track of the current position within the collection.
   - **Concrete Iterator**: These are specific implementations of the Iterator interface for different types of collections.
   - **Aggregate**: This is the collection of objects that you want to iterate over. It provides a method to create an iterator.
   - **Concrete Aggregate**: These are specific implementations of the Aggregate interface, representing different types of collections.
3. **How it works**:
   - The client (code that wants to iterate through the collection) requests an iterator from the aggregate.
   - The iterator keeps track of the current position within the collection and provides methods like `next()` to access elements sequentially and `hasNext()` to check if there are more elements.
   - The client can then use the iterator to traverse through the collection without knowing its internal details.
4. **Benefits**:

- **Separation of Concerns**: It separates the traversal logic from the actual collection, making code more modular.
- **Uniform Access**: Regardless of the type of collection (array, list, tree, etc.), you can use the same iterator interface to access elements.
- **Simplifies Client Code**: Clients don't need to know the specifics of how to access elements in a collection, making code more maintainable.

5. **Example**: Imagine you have a music player with a playlist (collection). The iterator allows you to go through the songs in the playlist one by one without needing to understand how the playlist is implemented.
6. **Common Usage**: Iterator pattern is commonly used in programming languages for iterating over collections like arrays, lists, and trees. It's also used in database query results, where you iterate over rows of data.
7. **Drawbacks**: In some cases, implementing custom iterators for different types of collections can add complexity. However, many programming languages provide built-in support for iterators.

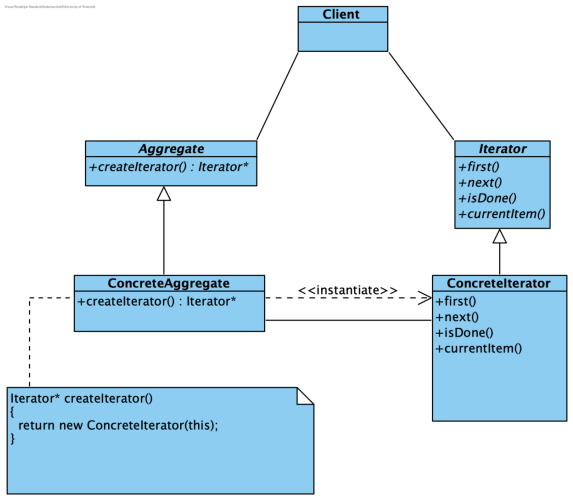
In summary, the Iterator Design Pattern simplifies how you traverse and access elements in a collection, making your code more flexible and maintainable while hiding the underlying collection details from the client code.

- **Classification**: Behavioural
- **Intent**: Provide a way to access the elements of an aggregate object sequentially without exposing its underlaying representation

Structure

Participants

- **Iterator**:
  - Defines an interface for accessing and traversing elements



> seperation of concerns:
> seperation of aggregate - how aggregate behaves from how it is traversed

- **Concrete Iterator**:
  - Implements the iterator interface
  - Keeps track of the current position in the traversal of the aggregate
- **Aggregate**:
  - Defines an interface for creating Iterator object
- **Concrete Aggregate**:
  - Implements Iterator creation interface to return an instance of proper concrete iterator

Related patterns

- **Factory method**: Both use subclass to decide what object should be created
- **Memento**: Often used in conjunction with iterator to capture the state of the aggregate. Memento is stored inside the iterator and is used for traversing aggregate
- **Adapter**: Both provide interface through which operations are performed.
- **Composite**: Recursive structures such as composites usually need iterators to traverse them sequentially

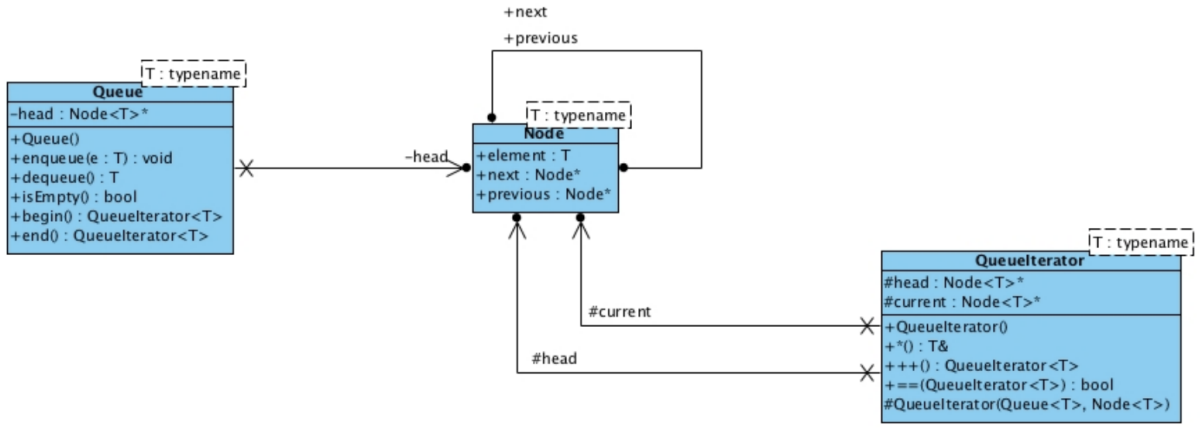
Example

Iterator in previous Observer example:

```
// Create the container
vector < PitCrew * > pitcrew;
```

```
...
// Insert elements into the container
pitcrew.push_back(person);
...
// Create an iterator
vector < PitCrew * > ::iterator it = pitcrew.begin();
// Iterate through the container
while ((it != pitcrew.end()) && (!found)) {
  if ( * it == person) {
    found = true;
    pitcrew.erase(it);
  }
  ++it;
}
// OR
vector < PitCrew * > ::iterator it = pitcrew.begin();
for (it = pitcrew.begin(); it != pitcrew.end(); ++it) {
  ( * it) -> update();
}
```

Queue iterator:




```
# ifndef _NODE_H
# define _NODE_H
template < typename T >
  class Node {
    public: T element;
    Node * next;
    Node * previous;
  };
# endif
```

```
# ifndef _QUEUE_H
# define _QUEUE_H
```

```
# include " Node . h"
# include " QueueIterator .h "
template < typename T >
  class Queue {
    friend class QueueIterator < T > ;
    public:
      Queue();
    void enqueue(T e);
    T dequeue();
    bool isEmpty();
    QueueIterator < T > begin();
    QueueIterator < T > end();
    private:
      Node < T > * head;
  };
# include " Queue . cpp " // Necessary - working with templates
# endif
```

```
# ifndef _QUEUE_C
# define _QUEUE_C
# include " Node . h"
# include " Queue .h"
# include " QueueIterator .h "
template < typename T >
  Queue < T > ::Queue() {
    head = 0;
  }
...
template < typename T >
  void Queue < T > ::enqueue(T e) {
    Node < T > * n = new Node < T > ();
    n -> element = e;
    if (isEmpty()) {
      n -> next = n;
      n -> previous = n;
    } else {
      n -> next = head;
      n -> previous = head -> previous;
      head -> previous -> next = n;
      head -> previous = n;
    }
    head = n;
  }
...
template < typename T >
  T Queue < T > ::dequeue() {
    if (isEmpty())
      return 0;
    else if (head -> previous == head) {
      Node < T > * tmp = head;
      head = 0;
      return tmp -> element;
    } else {
```

```
      Node < T > * tmp = head -> previous;
      head -> previous = head -> previous -> previous;
      head -> previous -> next = head;
      return tmp -> element;
    }
  }
...
template < typename T >
  bool Queue < T > ::isEmpty() {
    return head == 0;
  }
template < typename T >
  QueueIterator < T > Queue < T > ::begin() {
    return QueueIterator < T > ( * this, head);
  }
template < typename T >
  QueueIterator < T > Queue < T > ::end() {
    return QueueIterator < T > ( * this, head -> previous);
  }
# endif
```

```
# ifndef _QUEUEITERATOR_H
# define _QUEUEITERATOR_H
template < typename T >
  class Queue;
template < typename T >
  class Node;
template < typename T >
  class QueueIterator {
    friend class Queue < T > ;
    public:
      QueueIterator();
    T & operator * ();
    QueueIterator < T > operator++();
    bool operator == (const QueueIterator < T > & ) const;
    protected:
      QueueIterator(const Queue < T > & , Node < T > * );
    Node < T > * head;
    Node < T > * current;
  };
# include " QueueIterator . cpp "
# endif
```

```
# ifndef _QUEUEITERATOR_C
# define _QUEUEITERATOR_C
# include < iostream >
  using namespace std;
# include " QueueIterator .h "
# include " Queue .h"
# include " Node . h"
template < typename T >
```

```
  QueueIterator < T > ::QueueIterator(): head(0), current(0) {}
template < typename T >
  QueueIterator < T > ::QueueIterator(const Queue < T > & source,
    Node < T > * p): head(source.head), current(p) {}
template < typename T >
  T & QueueIterator < T > ::operator * () {
    return current -> element;
  }
template < typename T >
  QueueIterator < T > QueueIterator < T > ::operator++() {
    if (this != nullptr)
      this -> current = this -> current -> next;
    return * this;
  }
template < typename T >
  bool QueueIterator < T > ::operator == (const QueueIterator < T > & rhs)
const {
  return current == rhs.current;
}
# endif
```

- Mediator -

The Mediator Design Pattern is a behavioral design pattern used to centralize complex communication and coordination between multiple objects. Here's an explanation in simpler terms:

1. **Purpose**: The Mediator pattern is used to reduce the direct connections and dependencies between objects, promoting loose coupling. It allows objects to communicate with each other through a central mediator rather than directly.
2. **Participants**:
   - **Mediator**: This is an interface or a class that defines the methods for communication between objects. It keeps references to all participating objects.
   - **Concrete Mediator**: This is a specific implementation of the Mediator interface. It manages and coordinates the interactions between objects.

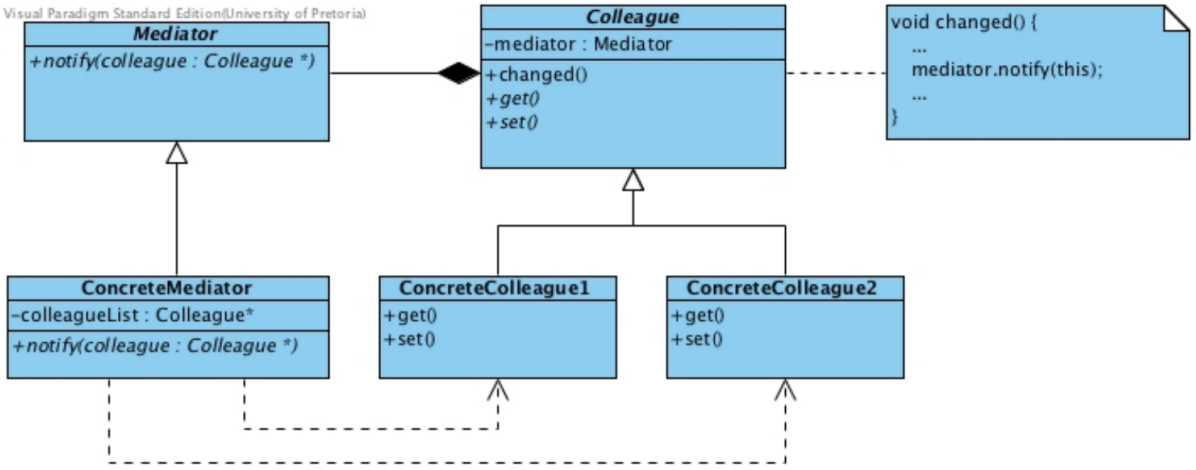
- **Colleague**: These are the objects that need to communicate with each other but do so through the Mediator. They are often unaware of each other.

3. **How it works**:
   - Colleague objects don't communicate directly with each other. Instead, they send messages or notifications to the Mediator.
   - The Mediator receives these messages and decides how to route or handle them, which may involve invoking methods on other colleagues.
   - Colleagues only need to know about the Mediator interface, not each other. This reduces dependencies and promotes a more maintainable and flexible system.
4. **Benefits**:
   - **Decoupling**: It reduces the dependencies between objects, making the system more modular and easier to maintain.
   - **Centralized Control**: Complex communication logic is centralized in the Mediator, making it easier to manage and modify.
   - **Reusability**: Mediators can be reused for different scenarios with different sets of colleagues.
5. **Example**: Think of an air traffic control system as a Mediator. Various aircraft (colleagues) need to communicate and coordinate their movements, but they don't talk directly to each other. Instead, they communicate through the air traffic controller (the Mediator).
6. **Common Usage**: The Mediator pattern is commonly used in GUI frameworks, chat applications, and systems where multiple components need to interact without knowing the specifics of each other.
7. **Drawbacks**: Introducing a Mediator can add some complexity to the system, and it might be overkill for simpler applications. It's best suited for scenarios with many interacting objects.

In summary, the Mediator Design Pattern provides a way to centralize and manage communication between objects, reducing dependencies and promoting a more organized and flexible design.

- **Classification**: Behavioural
- **Intent**: Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary the interaction independently.

Structure



Participants

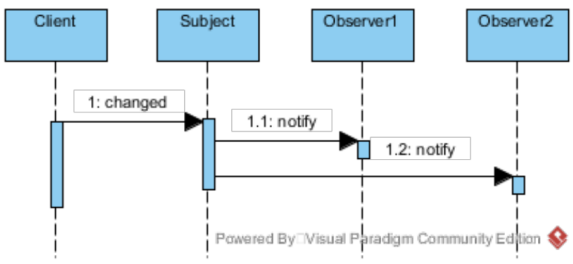
- **Mediator**:
  - Defines interface for communicating with Colleague objects
- **ConcreteMediator**:
  - Implements cooperative behaviour by coordinating Colleague objects
  - Knows and mantains colleagues

Related patterns

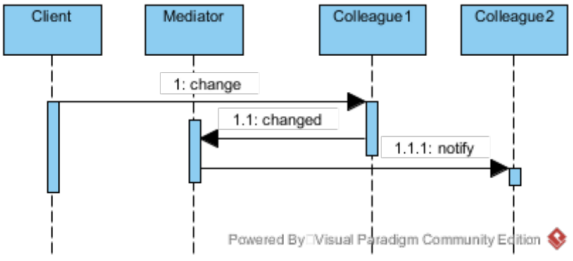
- **Facade**: Undirectional behaviour - between Facade interface to the subsystem objects, but not back. Mediator allows multidirectional communication.
- **Observer**: Colleagues can use Observer pattern to communicate with mediator.
  - Mediator extends Observer pattern:

- **Colleagues**:
  - Each colleague class knows its Mediator object
  - Each colleague communicates with its mediator whenever it would have otherwise communicated with another object

- Observer registers observers that get updated whenever subject changes
- Mediator reguisters colleagues that get updated whenever one of the other colleagues notifies mediator of update



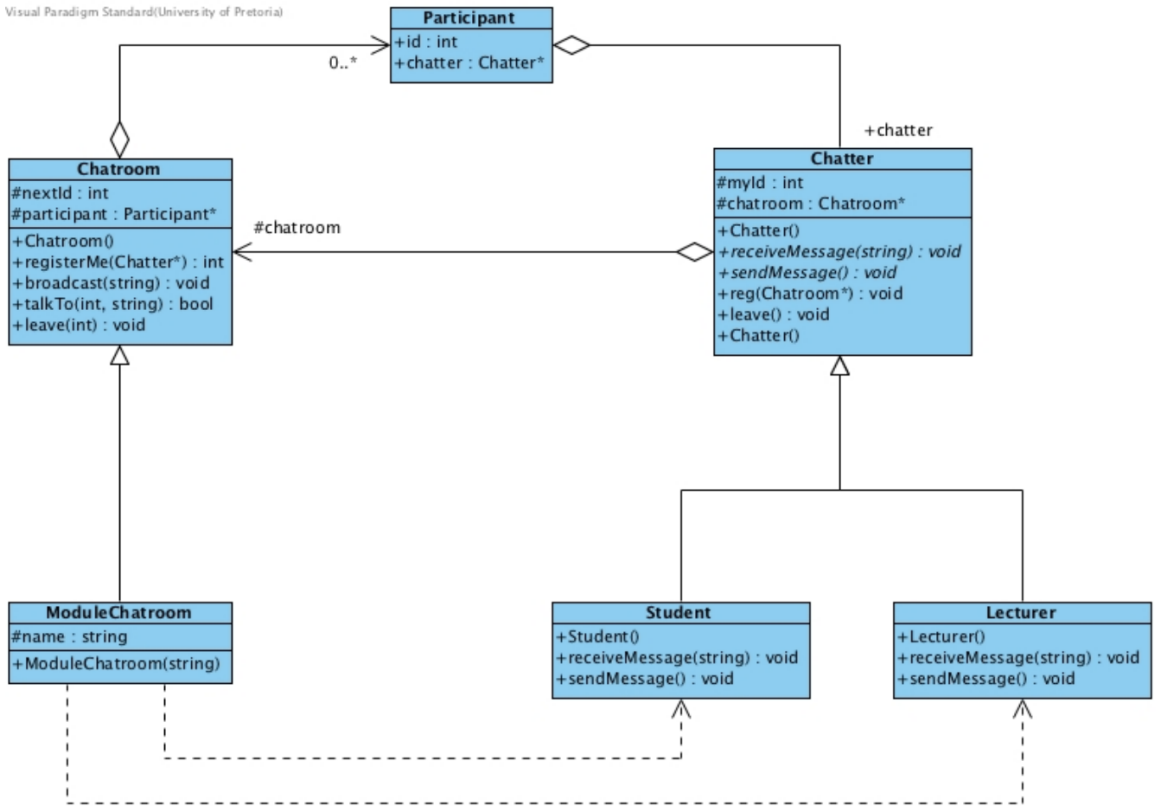
Observer



Mediator

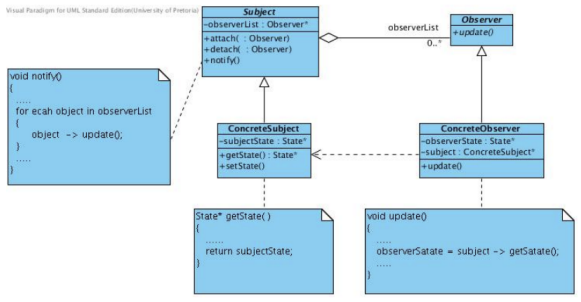
Example

Chatroom example:

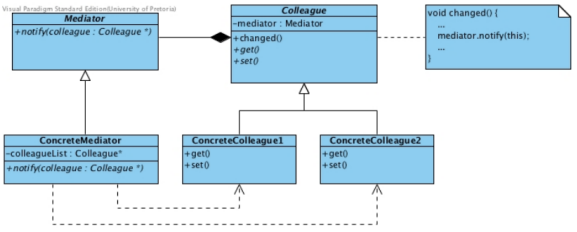


- **Mediator**: Chatroom
- **ConcreteMediator**: ModuleChatroom
- **Colleague**: Chatter
- **ConcreteColleague**: Student, Lecturer

Pitstop revisited:



Observer



Mediator

- Command -

The Command Design Pattern is a behavioral design pattern used to encapsulate a request as an object, thereby allowing for parameterization of clients with queues, requests, and operations.

1. **Purpose**: The Command pattern helps in decoupling the sender (client) of a request from the receiver (the object that performs the action). It turns a request into a standalone object that contains all the information needed to perform an action.
2. **Participants**:
    - **Command**: This is an interface or a class that defines the execution method. It represents a specific action to be taken.
    - **Concrete Command**: These are specific implementations of the Command interface, each encapsulating a particular action along with the necessary parameters.
    - **Invoker**: This is the object that sends the request to the Command. It does not need to know how the command is executed.
    - **Receiver**: This is the object that performs the actual action when the command is executed. It knows how to carry out the command.
3. **How it works**:
    - The client (Invoker) creates a Command object and associates it with a Receiver.
    - The Command object encapsulates the request, including all the necessary information and parameters.
    - When the client wants the request to be executed, it simply calls the `execute` method on the Command.
    - The Command then invokes the corresponding method on the Receiver, which carries out the action.
4. **Benefits**:

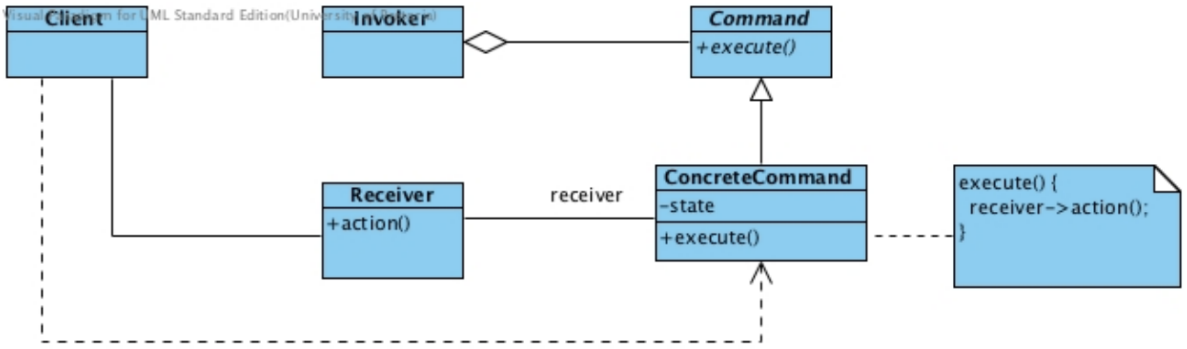
- **Decoupling**: It separates the sender and receiver, reducing dependencies between them.
- **Flexibility**: You can easily add new commands without modifying existing code.
- **Undo/Redo**: Commands can be stored for undo/redo functionality.

5. **Example**: Think of a remote control for a TV. The remote control (Invoker) has buttons for different commands like "Turn On," "Turn Off," and "Change Channel." Each button is associated with a specific Command that knows how to perform that action on the TV (Receiver).
6. **Common Usage**: The Command pattern is commonly used in GUI applications for actions like button clicks, in multi-level undo/redo systems, and in queuing and scheduling of tasks.
7. **Drawbacks**: In some cases, it can lead to a proliferation of command classes if you have many different actions to encapsulate. However, this can also promote a more organized codebase.

In summary, the Command Design Pattern encapsulates requests as objects, allowing for flexible and decoupled execution of actions in a way that promotes maintainability and extensibility.

- **Classification**: Behavioural
- **Intent**: Encapsulate a request as an object, thereby letting you parameterise clients with different requests, queue or log requests, and support undoable operations

Structure



Participants

- **Command**:
  - Declares interface for executing an operation
- **ConcreteCommand**:
  - Defines binding between a receiver object and action
  - Implements execute() by invoking corresponding operation(s) on Receiver
- **Client**
- **Invoker**:
  - Asks command to carry out request
- **Receiver**:
  - Knows how to perform operations associated with carrying out request
  - Any class may serve as receiver

Related patterns

- **Chain of Responsibility**: Can use command to represent requests as objects
- **Composite**: Can be used to implement MacroCommands
- **Memento**: Can keep state of the command required to undo its effect
- **Prototype**: A command that must be copied before being placed on history list acts as Prototype

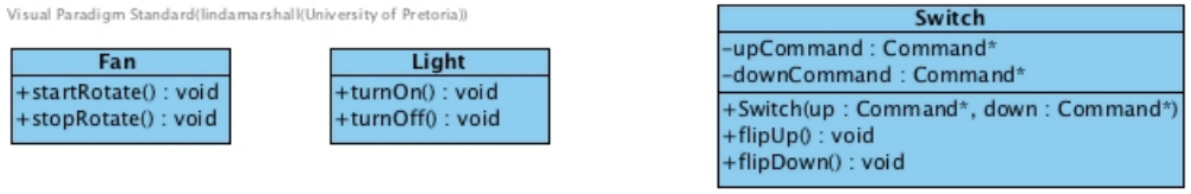
Example

Ceiling fan:


```
Light* testLight = new Light();
Fan* testFan = new Fan();
testLight -> turnOn();
testLight -> turnOff();
testFan -> startRotate();
testFan -> stopRotate();
```

We would like to use instruction to:

- turn light on/start rotating fan - `flipup` in `Switch` class
- turn light off/stop rotating fan - `flipdown` in `Switch` class



- **Invoker**: `Switch`
- **Receiver**: `Fan` and `Light`

```
class Fan {
  public:
    void startRotate() {
      cout << "Fan is rotating" << endl;
    }
    void stopRotate() {
      cout << "Fan is not rotating" << endl;
    }
};

class Light {
  public:
    void turnOn() {
      cout << "Light is on" << endl;
    }
    void turnOff() {
      cout << "Light is off" << endl;
    }
};
```

```
class Switch {
  public:
    Switch(Command* up, Command* down) {
      upCOmmand = up;
      downCommand = down;
    }

    void flipUp() { upCommand->execute();};
    void flipDOwn() {downCommand->execute();};

  private:
    Command* upCommand;
    Command* downCommand;
};
```

```
class Command {
  public:
    virtual void execute() = 0;
};

class LightOnCommand : public Command {
  public:
    LightOnCommand (Light* L) {myLight = L;}
    void execute() {myLight->turnOn();}
  private:
    Light* myLight;
};

class LightOffCommand : public Command {
  public:
    LightOffCommand (Light* L) {myLight = L;}
    void execute() {myLight->turnOff();}
  private:
    Light* light;
};

class FanOnCommand : public Command {
  public:
    FanOnCOmmand (Fan* f) {myFan = f;}
    void execute() {myFab->startRotate();}
  private:
    Fan* myFan;
};

class FanOffCommand : public Command {
  public:
    FanOffCommand (Fan* f) {myFan = f;}
    void execute() {myFab->stopRotate();}
  private:
    Fan* myFan;
};
```

```
Light* testLight = new Light();
Fan* testFan = new Fan();

LightOnCOmmand* testLiOnCmnd = new LightOnCommand(testLight);
LightOffCommand* testLiOffCmd = new LightOffCommand(testLight);
FanOnCommand* testFanOnCmnd = new FanOnCommand(testFan);
FanOffCOmmand* testFaOffCmnd = new FanOffCOmmand(testFan);

Switch* lightSwitch = new Switch(testLiOnCmnd, testLiOffCmnd);
Switch* fanSwitch = new Switch(testFaOnCmnd, testFaOffCmnd);

lightSwitch->flipUp();
lightSwitch->flipDown();
fanSwitch->flipUp();
fanSwitch->flipDown();
```

- Activity diagrams -

- Often used to model business processes
- Model complex workflows by operations on objects
- Describe what actions need to take place and when they should occur
- Describe how activities are coordinated to provide a service

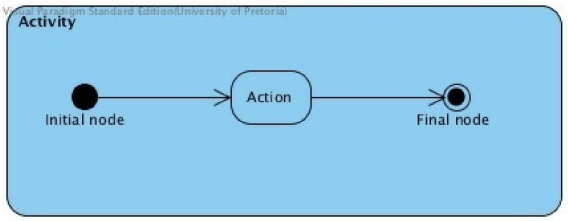
State diagram vs activity diagram

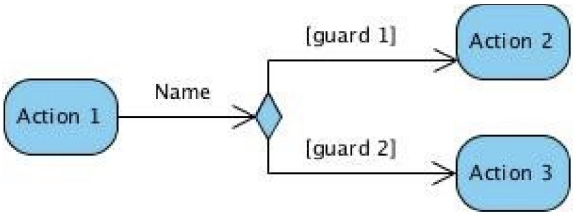
- **State diagram**: used to model state-dependent behaviour and conditions for transitions between states
- **Activity diagram**: used to model flow of actions and the order in which actions take place

Notation

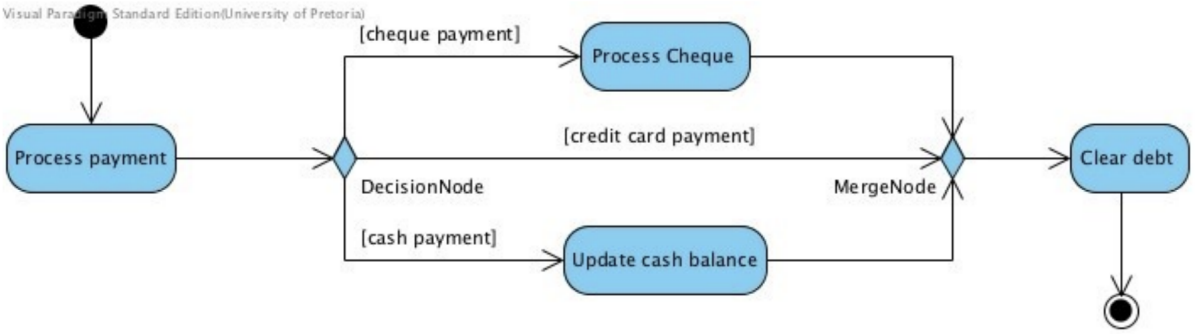
Start, action and end nodes

Activity edge and [guard condition]

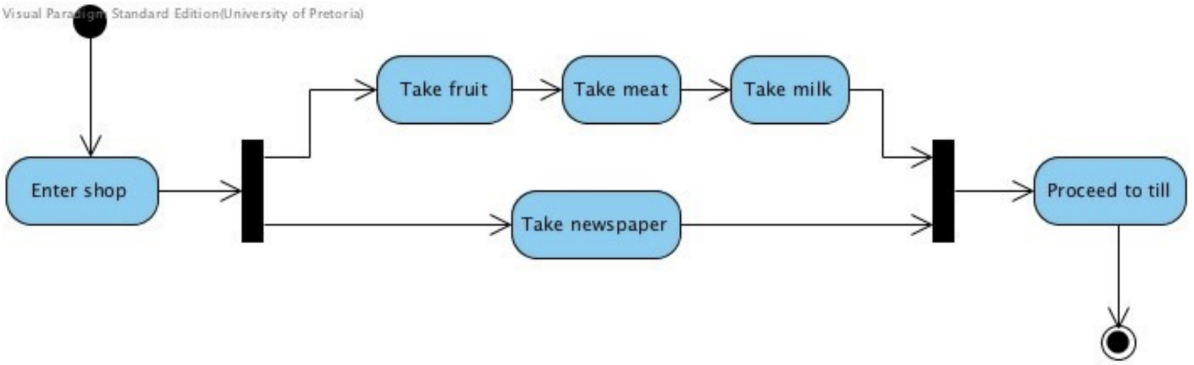




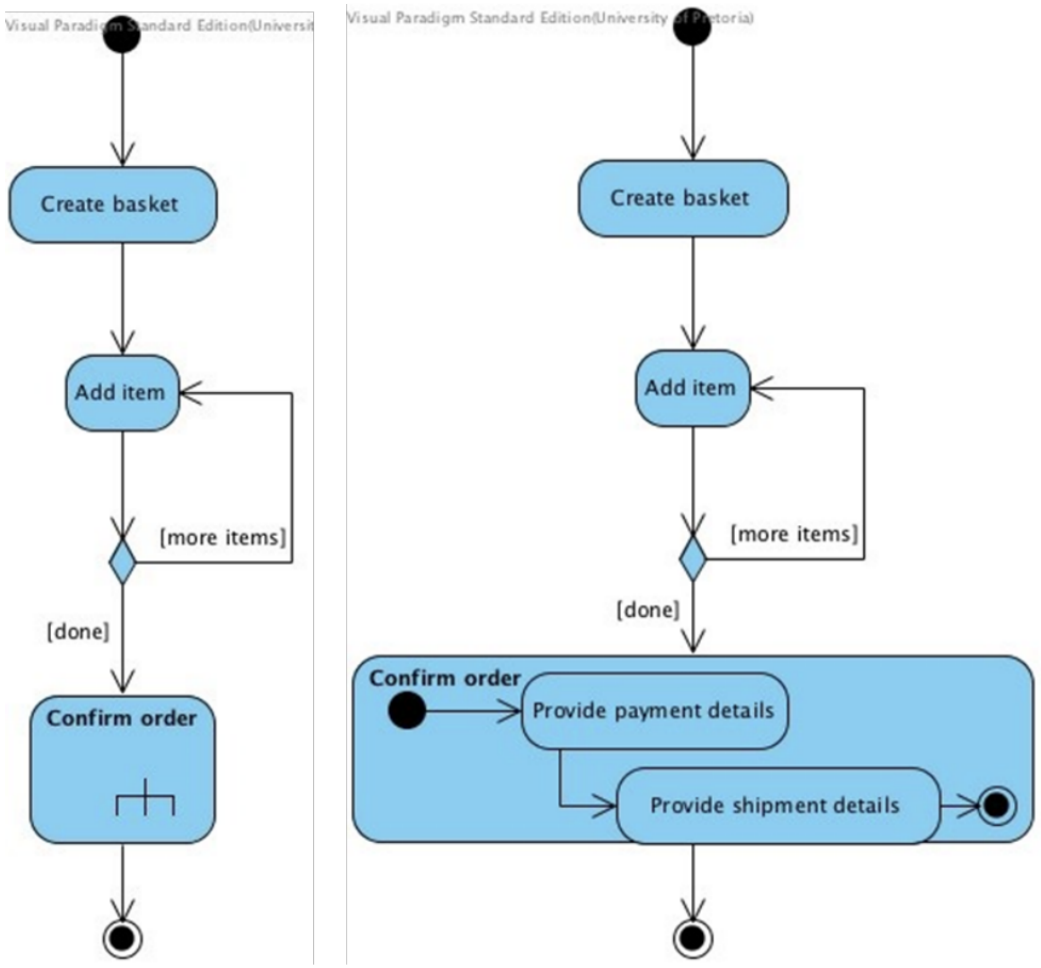
Alternate flows



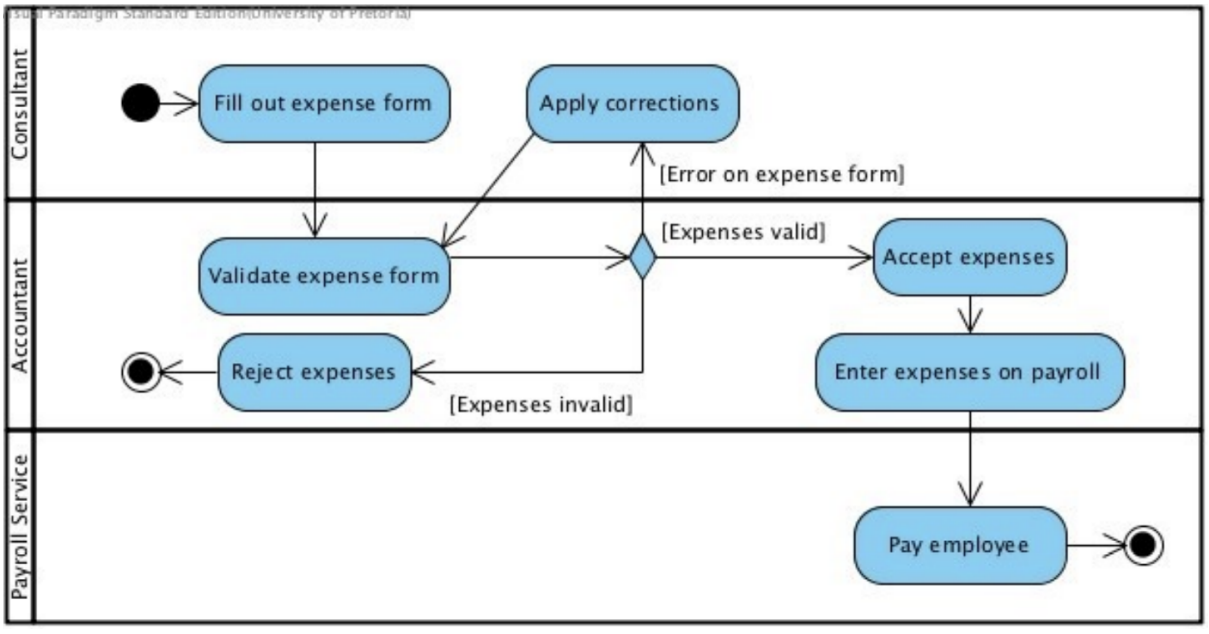
Parallel flows

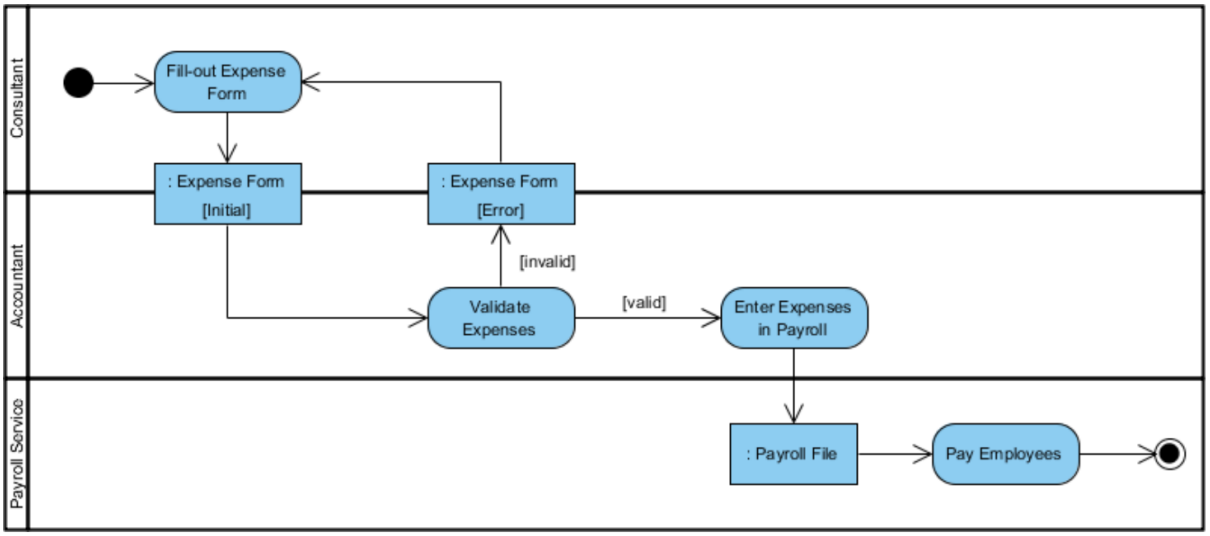


Composite activities



Swimlanes - used to convey which class is responsible for given activity





Example of finding and drinking beverage

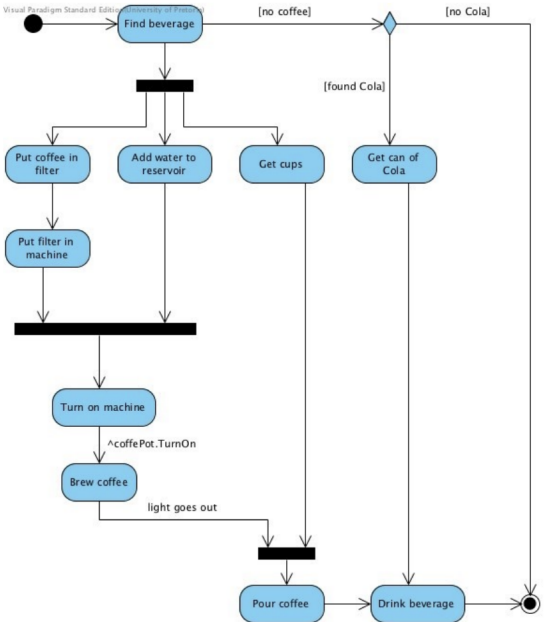
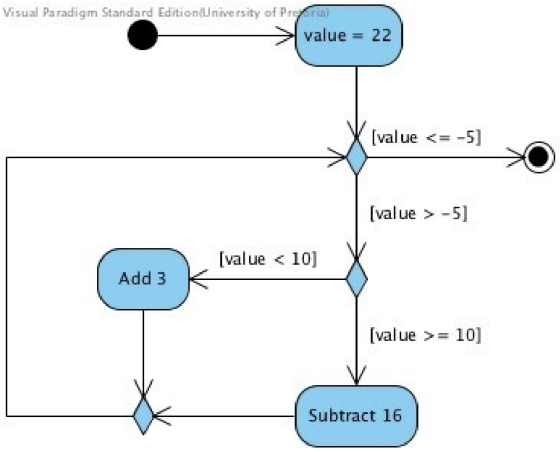


Diagram containing loop



- Software testing -

Define and implement service

- Interface

Failure of testing

- **Therac-25 bug**: Delivered to more than 100 time

- Service name → Class name
- Operation name → Function name
- Operation has preconditions/postconditions
- Exceptions that can be thrown

Characteristics

- Start from inside and work toward outside
  - Function level
  - Class level
  - Between classes
  - Between modules
  - Between systems
- Different techniques
- Testing is not the same as debugging

intended dose of radition to patients. Two died☠️

- **Ariane 5 rocket**: 37 seconds after launch destroyed, costing $370M. Error in 64-bit floating point to 16-bit signed conversion error.
- 1200 US Veterans wrongly informed they had fatal Lou Gehrig’s neurological disease
- **MIM-104 Patriot**: System clock drift resulting in failure to intercept incoming missile, killing 28 Americans💀😵
- **Sony BMG rootkit scandal**: Installing rootkits on Windows machines

Creating & destroying

- **Dichotomy**: a division or contrast between two things that are (or represented as) being opposed or entirely different.
- **Building** software - creating
- **Testing** software - destroying

Why test?

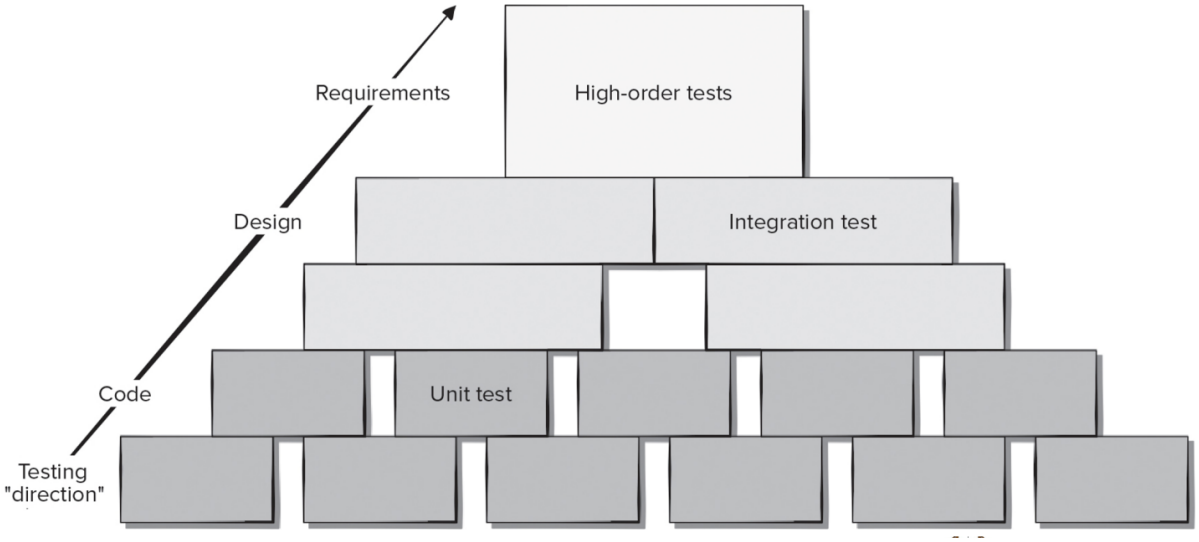
- Software Quality Assurance
- Ensure we build good and correct software

What test?

- Abstract solution
- Implemented/Realised solution

- Smallest to largest component

Steps to test



Types of testing

- **Unit testing**: Small units (class, functions)
- **Integration testing**: Does it all fit together and work?
- **Validation testing**: Requirements are checked against software
- **Regression testing**: Retesting components that may be affected by changes
- **System testing (e2e)**: Tested as one unit

Testing frameworks

- **Automated**
  - Microsoft Unit Testing Framework C++
  - Google Test
  - Boost. Test
  - CTest

Test design

- **Exhaustive testing**
- **Test smart**

    - Not feasible
    - Cost not worth it
    - Crucial modules
    - Error-prone modules

Test case design:

- Testing module interface
- Local data structures stored data mantains integrity
- Independent paths are tested
- Boundary conditions
- All error-handling paths

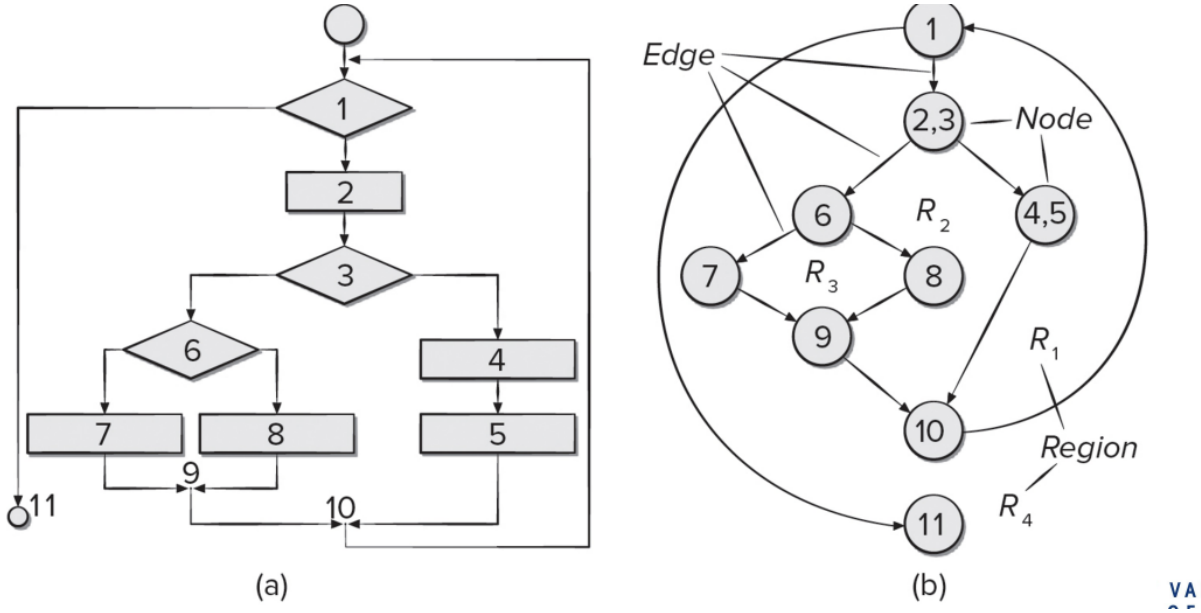
Traceability:

- Each test case should be traceable to preconditions, postconditions or exceptions
- Traceable back to API design
- Test failures often due to missing traceability, inconsistent test data, incomplete coverage

White-box testing

- Guarantee that all independent paths within a module have been excercised at least once
- Excercise all logical decisions on their true and false sides
- Execute all loops at their boundaries and within their operational bounds
- Excercise internal data structures to ensure validity

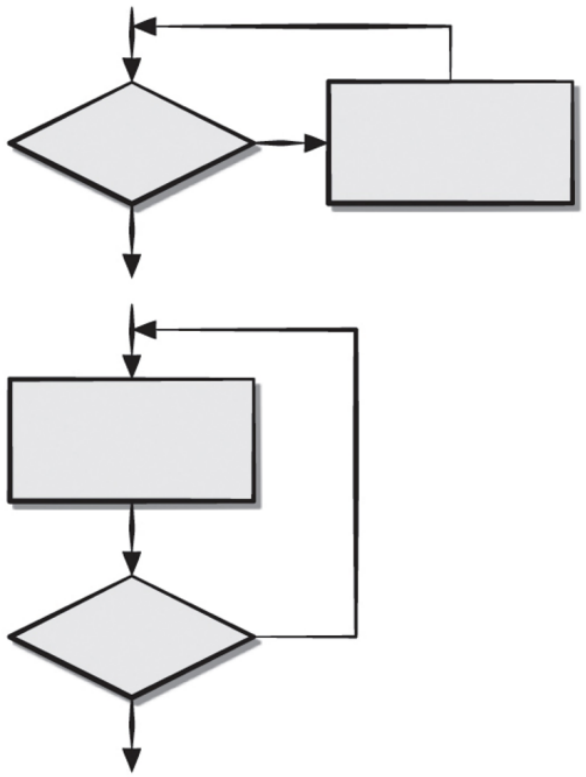
Basis Path Testing:



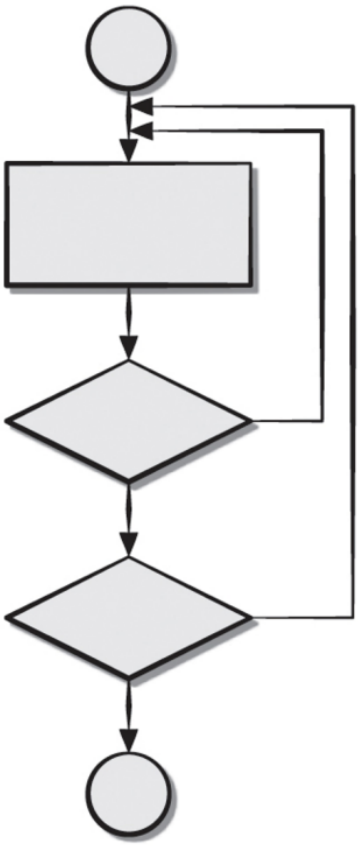
- Independent path: Any path through program that introduces **at least one** new set of processing statements or condition
- Path 1: 1-11
- Path 2: 1-2-3-4-5-10-1-11
- Path 3: 1-2-3-6-8-9-10-1-11
- Path 4: 1-2-3-6-7-9-10-1-11

Control Structure Testing:

- **Condition testing**: Method that excercises logical conditions
- **Data flow testing**: Selects test paths of program according to variables
- **Loop testing**: Focuses exclusively on validity of loop constructs



Simple loops



Nested loops

Simple loop testing:

- Skip loop entirely
- Only one pass through loop
- Two passes through loop
- m passes through loop where m < n
- n1, n, n+1 passes through loop