IP1

The focus of IP1 is on correctness and termination of simple imperative programs. To consider this we introduce the concepts of the invariant and the variant for loops, within the wider paradigm of Hoare logic. Where $\{P\}\ \texttt{command}\ \{Q\}$ is the proposition that given P is true about the state, then on the execution of `command`, Q is true, we get the following basic facts:

$$\frac{\{B \wedge P\}\ \texttt{c}_1\ \{Q\} \wedge \{\neg B \wedge P\}\ \texttt{c}_2\ \{Q\}}{\{P\}\ \texttt{if}\ B\ \texttt{then}\ \texttt{c}_1\ \texttt{else}\ \texttt{c}_2\ \{Q\}} \quad \text{correctness of } \texttt{if}$$

$$\frac{\{P \wedge B\}\ \texttt{c}\ \{P\}}{\{P\}\ \texttt{while}\ B\ \texttt{do}\ \texttt{c}\ \{\neg B \wedge P\}} \quad \text{termination with invariant } P$$

The notion of the invariant uses the second fact: that provided we can guarantee both $\{P \wedge B\}\ \texttt{c}\ \{P\}$ and termination of $\texttt{while}\ B\ \texttt{do}\ \texttt{c}$ given P, then we terminate with P and $\neg B$. If we can prove the correct result from this termination, then the program is correct.

To prove termination, we use the notion of the variant. If we have $\{P \wedge B \wedge t = t_0 \in \mathbb{N}\}\ \texttt{c}\ \{P \wedge t < t_0 \wedge t \in \mathbb{N}\}$, thus $t = 0 \Rightarrow \neg B$ so we have termination within a finite number of steps.

Binary Search

```
// Pre: For all 0 <= i < j < n, arr(i) <= arr(j)
// Post: @returns i in [0..n] such that arr[0..i) < x <= arr[i..n)
def binarySearch(arr: Array[Int], x: Int): Int =
  var l = 0; var r = arr.length
  // I: 0 <= l <= r <= n
  //    && arr[0..l) < x <= arr[r..n)
  // V: r - l
  while l < r do
    val m = l + (r - l) / 2 // l - 1 < floor((r + l) / 2) < r
                            // so l <= m < r
    if arr(m) < x then
      l = m+1 // l <= r && arr[0..l) < x <= arr[r..n)
              // && r' - l' = r - m - 1 <= r - l - 1 < r - l
    else
      r = m   // l <= r && arr[0..l) < x <= arr[r..n)
              // && r' - l' = m - l < r - l
  l
```

IP2

The focus of IP2 is on datatype stability. For a specified datatype, we have both an interface and an implementation.

For each interface we describe it by its abstract state (e.g. an object S implementing the Set[T] interface satisfies $S : \mathcal{P}(T)$), its initial state (e.g. $S = \varnothing$), and the pre- and post-conditions of each method described in the interface.

For each implementation we prove its correctness by describing its abstraction function f from the set of states which its attributes may place the object in, to the abstract set the object represents as described in the interface. Strictly this function ought to be surjective, as otherwise there are abstract objects the implementation is incapable of representing. We also describe a datatype invariant D, a proposition about the attributes of the object which is true upon initialisation, and for each possible call $\texttt{c}$ which could be required of the object, $\{D\}\ \texttt{c}\ \{D\}$.

IP3

Our focus in IP3 shifts towards correctness in larger program environments.

The first idea of object orientation is the notion of object identity - that only the object itself is identical to itself. Different instantiations create different objects.

The second idea is encapsulation: that the internal implementation of a component will not be accessible via its external interface. This allows debugging to occur more simply, as any problems can only occur via the failure of a distinct object of its internal means, rather than being broken from outside.

The principle of Demeter is the principle that an object should only interact with methods within itself or created by itself, rather than calling methods on other objects.

Object orientation

Inheritance

Inheritance is a feature by which classes may automatically have access to methods defined for another class (with the capacity to override them if desired), and thus may be used as if they are that class. Additionally, they may enjoy their own methods, although only when typed as their own class rather than that of the parent.

While inheritance is useful for providing structure in static, short-lived environments, as it allows us to avoid repeating code blocks to achieve similar behaviour between objects, it introduces a significant problem to the coherency of the program as a whole. This is that child classes override the methods which the base class may itself call, introducing customisation points and thus potentially leading to unintended behaviour. We can however avoid this somewhat by replacing customisation points with private methods. **Is this still a problem?**

Composition

Composition is a different pattern of coupling objects together whereby an object becomes an attribute of another, rather than inherently linking the classes. This allows for the outer class to serve as an interface to the inner class, while implementing additional functionality on top of it.

The main drawback of composition is that we can no longer type the class as the class it contains. Thus we cannot deal with it in a polymorphic manner. At the same time the only coupling occurring is in one dimension – the outer class relies on the inner class to behave as expected, while the inner class may be altered however one might like without forgoing its specification.

Design Patterns

Adapter

An Adapter is made up of a target, the client, the adaptee, and the adapter. The target is the interface we intend to provide, the client the codebase which expects instances the type of the target, the adaptee the current provision, and the adapter the presentation of the adaptee as the type of the target.

Iterator

An iterator is a method of getting access to the elements of a collection without being able to change it or needing to store it in any way.

```
trait Iterator[+T] {
  def hasNext(): Boolean
  def next(): T
}
```

Observer

An Observer pattern is composed of four components:

- **Observer**: An interface with a method to be called when changes occur.
- **Subject**: A class for notifying observers of changes.
- **Concrete Subject**: The class to be observed, responsible for notifying the observer.
- **Concrete Observer**: A concrete implementation of the observer.

We can write the subject and observer as follows:

```
trait Observable[T] { this: T => // Observable[T] must be mixed in with another
 trait.
  val observers = new ArrayBuffer[Observable.Observer[T]]

  def addObserver(observer: Observable.Observer[T]) {
    observers.append(observer)
  }

  def notifyObservers() {
    for (o <- observers) o.refresh(this)
  }
}

object Observable {
  trait Observer[T] {
    def refresh(subject: T)
  }
}
```

and then any concrete subject extends T with Observable[T] while any concrete observer extends Observable.Observer[T].

Model View Controller

The model-view-controller design pattern is an architecture for structuring GUI applications. It is composed of the model, the view (the code responsible for the presentation of the model to users), and the controller, which is responsible for processing user input and updating the model.

Façade

A façade is a pattern formed of a subsystem, a client, and an interface between the two (the façade class itself).

Factory

The factory pattern is a pattern whereby instead of creating objects by explicitly instantiating the class alongside its arguments, one creates a factory object with a method that abstracts the details of the constructor away from the user.

For the purpose of generality, we have an abstract factory class which defines a creation method, and each class to be created using it has an associated factory object inheriting it. Take below as an example:

```
trait Factory {
  def apply(filename: String): Reader
}

object ReaderFactory extends Factory {
  def apply(s: String): Reader = {
    fileExtension(s) match {
      case ".csv" => csvReader(s)
      case ".json" => jsonReader(s)
      case ".parquet" => parquetReader(s)
      case _ => throw new RuntimeException("Unknown file type")
    }
  }
}
```

Command

The command design pattern contains the command interface, typically containing the method execute returning Unit, a receiver which performs the command, the concrete commands implementing the interface, the client object creating commands, and the invoker object which runs the commands when appropriate.

Memento

A Memento is a pattern which captures the state of an originator object, allowing for the restoration of the object without exposing anything further. It ought not to be possible for the state to be accessed prior to its restoration.

Decorator

The Decorator pattern entails that instead of creating an explosion of subclasses, define a subclass meant to be "decorated". It is an example of an instance where inheritance is a very poor solution to the overall problem, as we desire to alter the behaviour of an object at runtime.

The components of a decorator are as follows:

- The component interface, for the objects being decorated.
- The concrete implementation of the component, inheriting the component interface.
- The base decorator, inheriting the component interface and storing a component object.
- The set of concrete decorators inheriting the base decorator.

To implement these, we write the base decorator's methods to imitate those of its wrapped component. Thus the concrete decorators may then implement theirs to call the super methods in between doing their own extra work.