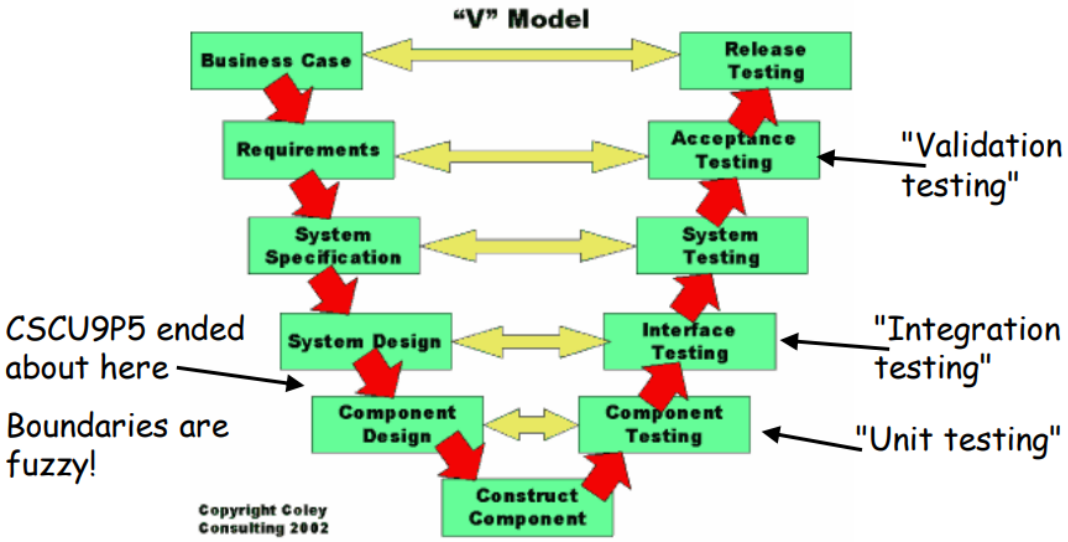
SOFTWARE IMPLEMENTATION

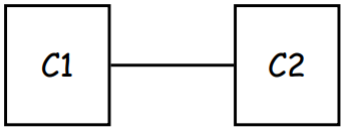
INTRO TO CSCU9P6

1. What is the **V model**? Draw a diagram of it.
   The V model is a view of the relationship between analysis, design, coding and testing.

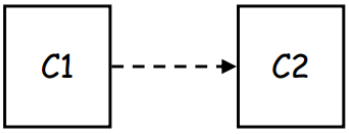


2. Describe some examples of **detailed design activities**.
   Reviewing/refining the object model, refactoring the object model, selecting an architectural framework, concretizing associations, and implementing classes.

3. What is involved in **refining associations**?
   An association is a conceptual relationship between instances of classes. As our understanding of the design improves, and as implementation progresses, we refine the association:
   - Maybe just dependency
   - Perhaps one class holds/refers to instances of the other
   - Decide whether it is conceptual aggregation or composition
   - Identify its navigability/directionality
   - Identify its multiplicity



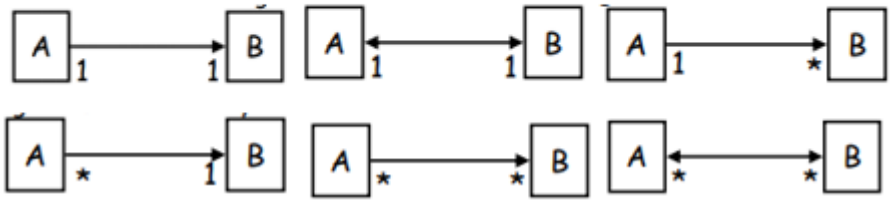
4. Describe how **dependencies** work.
   When one class has an operation or result from another class, or if an attribute in one needs to change if it changed in the other. For example, C1 depends on C2 if C1 has an operation parameter or result of type C2, or a local (temporary) variable of type C2, or C1 has no persistent "ownership" of, or access to, an object of class C2 –no attribute –but C1's code might need to change if C2 changes.



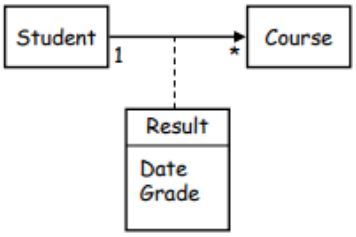
5. What is an **attribute-based association**?
   If an association is not a simple dependency, say if C1 has persistent ownership of or access to a C2; C1 may call on the services of C2.

6. Give some **examples** of associations we can have.
   Directionality - 1-1, bi-directional, 1-many,

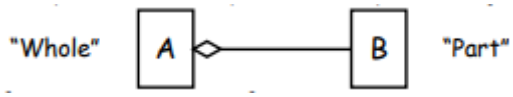
   many-1, many-many, many-many (multi directional), all shown below in this order.



7. What is an **association class**? How can we implement them?
   Information attached to an association.



8. What is **aggregation**, and how do we implement it?
   A conceptual hierarchical structure, such as book -> magazine. Often a 1-1 or 1-many association.

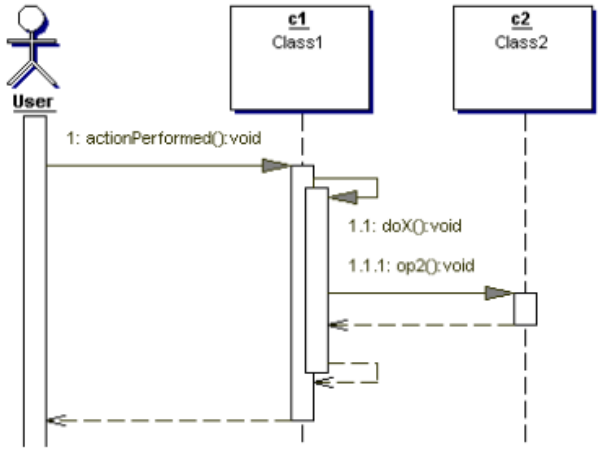


9. What is **composition**, and how do we implement it?
   Strong form of aggregation, the parts have no existence without the whole. B cannot exist without A. Like a chessboard, board -> square

USE CASE & SEQUENCE DIAGRAMS

1. What is a **use case**?
   A use case describes an interaction that an actor will have with the system.
2. What is a **sequence diagram**?
   A sequence diagram shows how an input “stimulus” from an actor causes interaction between several objects within the system.
3. Describe how **use cases** and **sequence diagrams** relate to one another, and how they are both implemented.
   Usually there may be one sequence diagram for each use case – sequence diagrams emphasise the time ordering of messages sent between collaborating objects.
   You cannot really implement a sequence diagram, as it is a guide for implementing different parts of the system usually scattered over a collection of classes.

4. How can we show **actions/boundary classes** in sequence diagrams?
   The initial message in a sequence diagram is (usually) from an actor to a boundary class object; could simply be a call of a public operation from some other part of the system, i.e. a diary/calendar sub-system might "prompt" the core system to carry out an action at certain times.

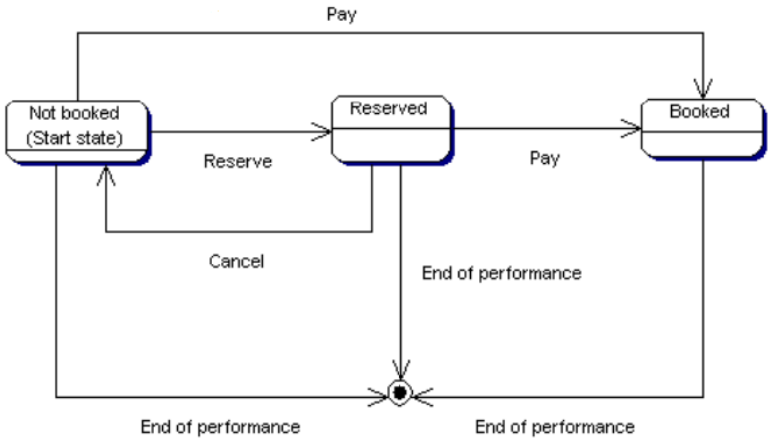


IMPLEMENTING STATE DIAGRAMS

1. What is a **state diagram**? How does it work?
   Tool to allow us to describe non-trivial internal changes to an object, in relation to external events, i.e. do this, do that…
   The initial state/start point is indicated by a black sot, and the end point by a black dot within a circle.
   Guards are used when there are conditions to be met; for example, pay could have a guard of [cash >= payment] before the state is changed.

2. Describe what is meant by the following terms:
   a. **Event name – the descriptive name of the event (the arrows)**
   b. **Event arguments – any arguments the event uses**
   c. **Guard condition – any prerequisites for the state change**

3. Draw an example of a **state diagram**.



4. What is an **implementation requirement**?
   Things such as having enough attributes to represent all the states needed, enough methods for different events/states, and so on.

5. If a state diagram has an **action** on **transition** from **start symbol** to an **initial state**, where is that action placed?
   In the class's constructor.

IMPLEMENTATION ISSUES – REFACTORING ON DESIGNS

1. What is meant by **refactoring**?
   Rewriting existing source code with the intent of improving its design rather than changing its internal behaviour.

2. Describe some examples of **low level** and **higher-level** refactoring operations.

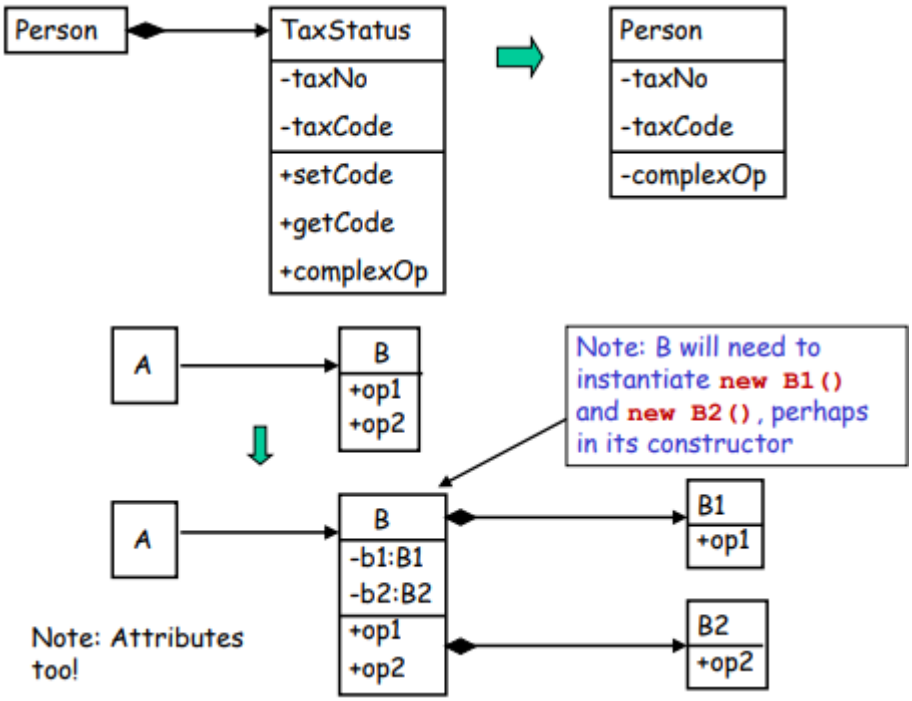
   Low level

   - "Pushing down" / "pulling up" an operation or attribute, from super to sub class and sub to super, respectively
   - Renaming a class, interface, attribute etc
   - Renaming a parameter
   - Extracting new super classes from single classes
   - Encapsulating an attribute: making it private and creating get/set methods.

   Higher level

   - If class B is a part of class A by aggregation or composition and class B is associated with no other classes, and are happy to lose B – may be able to remove B
   - Also, could split one class into two

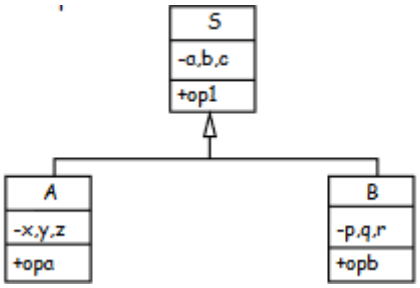
3. Why might we want to split a complex class into two or more **independent** classes?
   Could be better from an efficiency and implementation standpoint, but particularly a design standpoint as easier to understand. Splitting one class into two gives each class a more focused identity, for example.



4. How else can we use **class splitting**?

Could also retain B as a container for instances of the new classes; in this case B would keep the same public relations but their bodies would delegate the calls to appropriate instances of the new classes.

5. Why might we want to use refactoring to increase **reuse** within a system? How would we do this?
   Might introduce superclasses (possibly abstract) and have some related classes inherit from them to enable more beneficial code re-use.



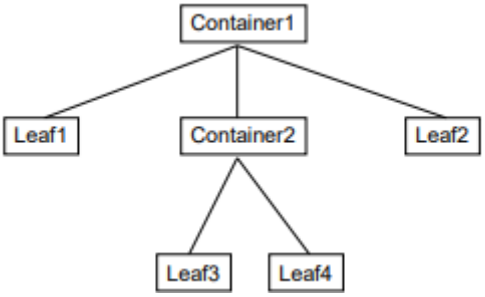
DESIGN PATTERNS

1. Why might we want to **reuse expressions**?
   If a problem is commonly occurring.

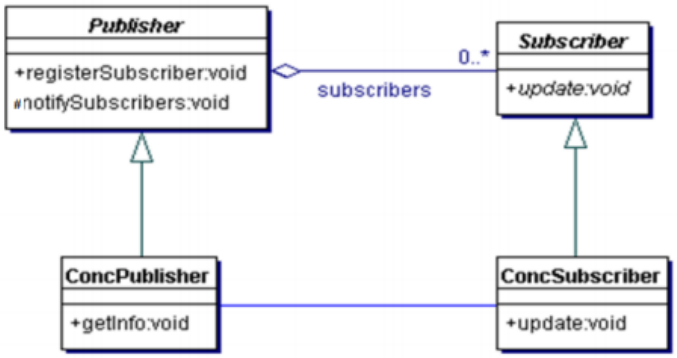
2. What is a **design pattern**?
   A re-usable solution to a commonly occurring problem.

3. Describe the **3 main types** of design pattern.
   - Creational – creating objects, e.g. Factory Method
   - Structural – class/objects structure, e.g. Adapter, Composite
   - Behavioural patterns – interaction, e.g. Publisher-Subscriber

4. What is the **composite pattern**? Describe how it works.
   The composite pattern uses a tree structure to manage hierarchies. It is the main architecture of the files and folders storage system.



5. What is the **publisher-subscriber pattern**? Describe how it works.
   A subscriber registers with a publisher, when the publisher has new information all subscribers are notified (used in assignment!)



MVC & IMPLEMENTATION ISSUES

1. What is the **MVC architecture**? Describe in detail how it is implemented.
   Used for highly interactive applications, based on the publisher-subscriber design pattern. Controllers receive inputs and send messages to the model. The views associated with controllers are notified whenever content in the model changes.
   This is exactly what the entire assignment was about so it is probably unlikely to be in the exam however is fairly simple to break down and explain. Its used in large systems that need to be constantly up to date and generally would work between different physical machines.

2. How do the Java library classes **Observer** and **Observable** work in terms of the MVC architecture?
   Observer is the Subscriber role; it is an interface – each view implements Observer. On the other hand, Observable is the Publisher role; it is a class – models extends Observable.

3. Why might we want each controller to appear in a **separate** window?
   Because you will probably want the system working across multiple machines, i.e. an air traffic control system.

SOFTWARE TESTING

1. Describe what is meant by the following types of tests:
   a. **Unit/component** test - To show whether a unit does or does not satisfy requirements and if its implementation does or does not match the design (Testing individual program components).
   b. **Integration** test - To show whether combinations of components are incorrect or inconsistent (Testing groups of components integrated to create a system or sub-system).
   c. **System** test - Concerns issues that can only be addressed by testing the entire system (Security, recovery, performance).
   d. **Validation/acceptance** test - Show if the system meets the client's requirements (directly related to use cases and scenarios).

2. What is the difference between **verification** and **validation**?
   Verification - Does it do what the specification asks it to do?
   Validation - Does it meet the specification of what the client wants it to do?

3. Why is **perfect testing** impossible?
   Too many possibilities; if program does something using just 2 integers, testing everything could take 16*10^12 seconds…

4. Describe the **testing process** involved in the 4 types of tests mentioned above.
    - Component/unit testing
        - Testing of individual program components
        - Usually the responsibility of the component developer (except sometimes for critical systems)
        - Tests derived from component descriptions/specifications
    - Integration testing
        - Testing of groups of components integrated to create a system or sub-system
        - Often the responsibility of an independent testing team
        - Tests based on a system specification

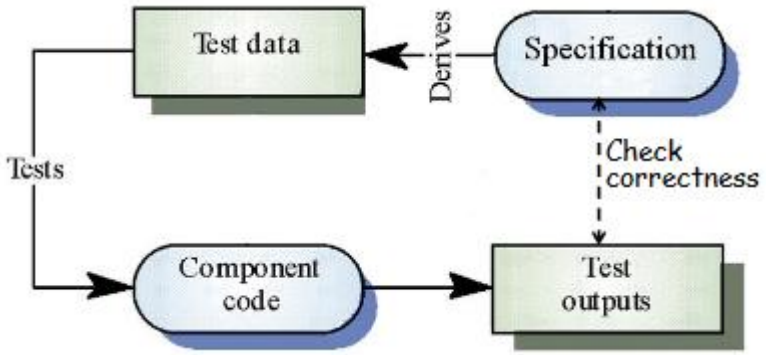
5. Describe what is involved in the **formality of testing**.
    - Inputs must be devised/prepared
    - Outcomes must be predicted
    - Test designs must be documented
    - Test must be executed
    - Results must be observed and recorded
    - Results must be compared with predictions

6. Why should we bother with **documentation**?
   It is a vital part of the formal process; also helps you to re-use previous work or understand what you were doing if you go back at a later stage.

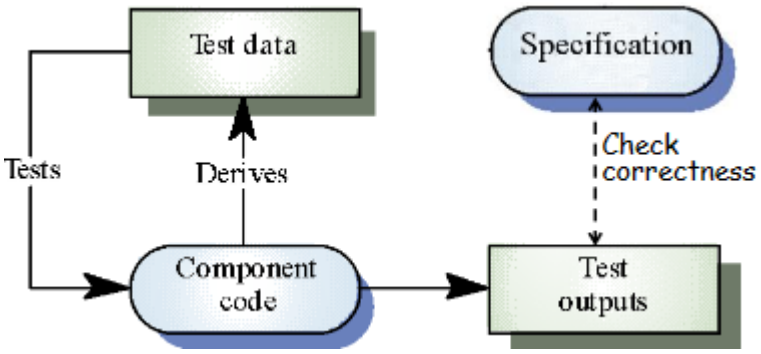
7. What is the difference between **test data** and **test cases**?
   Test data are inputs which have been devised to test the system.
   Test cases are a record of the tests chosen to test the system, with rationale, data inputs and predicted outputs.

8. Describe the difference between **black box** and **white box** testing.
    - Black box testing – knowing the expected functions, design tests to check whether each function behaves as expected
    - White box testing (structural testing) – knowing the internal operation of the code, design tests to exercise components, checking whether the overall behaviour is as expected

9. Draw a simple diagram of each in terms of **how** testing works on them.
    - Black box testing



    - White box testing



10. Can unit testing be both **white** and **black box testing**?
    Yes; tests individual system components (classes, methods). Uses stubs and/or drivers.
    - A stub is an empty or dummy implementation of a component that respects the public interface
    - A driver organises the tests – e.g. a JUnit test method

    The driver instantiates the component under test and calls its methods – supplying test data, gathering and reporting the results.

DEBUGGING

1. Describe the steps involved in the **general debugging process**.
    - Locate the fault
    - Design a repair
    - Carry out the repair
    - Re-test the program
    - Document the fault and the repair

2. Why is **retesting** the program important?
    - To check whether the repair worked
    - To check whether further faults have been introduced
        - In the repaired code
        - Or in code dependent on the repaired model

3. Describe some approaches to locating a **fault**.
    - Manually tracing the faulty test case through the code
    - Designing and running extra tests to localise/characterise the fault
    - Instrumenting the code with diagnostic statements
    - Using an interactive debugger to monitor the code

4. Describe some **diagnostic statements** we can use.
    System.out.println() or equivalent statements.

5. What is **interactive debugging**? How does it work?
    Interactive debuggers allow us to run the actual compiled code for a program with the ability to start and stop the program and monitor variable values at any time. For it to be useful the compiled code must retain information from the source code such as links to source code filenames/line numbers and the symbolic names for variables, methods, classes etc. used in the source code.
    Allows the interactive debugger to report a program's state to us in source code terms instead of just RAM addresses.

Typically allow setting breakpoints, stepping by single statements, inspecting variable values, etc.

6. What is a **breakpoint**? How do they work?
   Tells the execution to stop at that point so that debugging can take place.

7. Describe the **JVM scheme**.
   - Programs compile to bytecode
   - JVM is a hybrid interpreter / adaptive compiler- "hotspots" identified may be compiled to native code
   - JVM can be launched in a debugging mode in which it interprets the bytecode as normal, but "listens" for commands from a separate debugging program (which could even be on a remote computer!) **Socket based communication**
   - JVM can be instructed to stop at breakpoints, fetch variable values, etc.
   - JVM can stop interpretation of the bytecode at nominated locations -based on source code line numbers

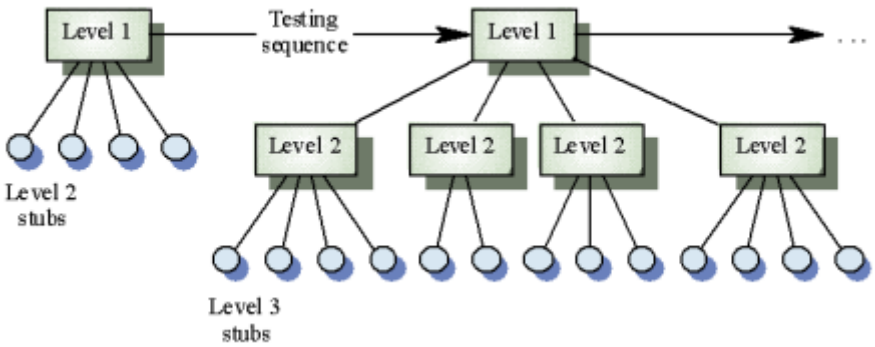
INTEGRATION TESTING

1. What is **integration testing**?
   Testing subsystems of integrated groups, building up gradually to test the entire system; uses stubs and drivers due to the gradual nature of the process.

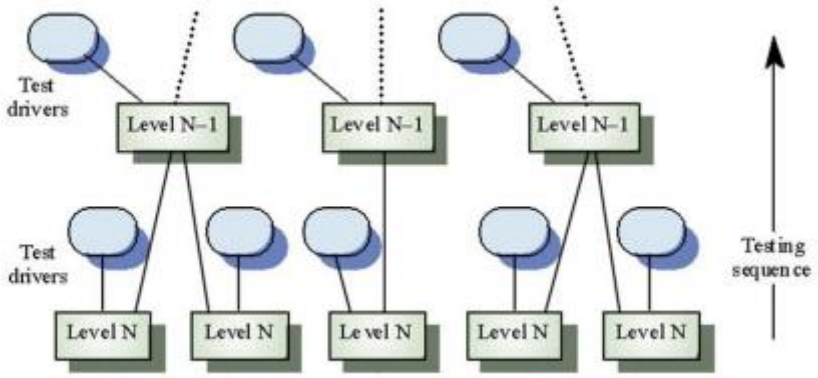
2. Should integration testing be **black** or **white box testing**?
   Should always be black box testing as the internal structure is too complex for white box.

3. What is the main **difficulty** with **integration testing**? How can we address this?
   Localising errors is the main difficulty – can be addressed through careful incremental integration testing.

4. Describe what is meant by **top-down** and **bottom-up** testing and draw a simple diagram of how both works.
   - Top-down testing – start with the high level, main components and work downward. Test harness lower level components and replace them with stubs.



   - Bottom-up testing – aggregate lower level individual components until eventually the entire system is covered. Test harness higher level components and replace them with drivers.



5. What is meant by **alpha** and **beta testing**?
    - Alpha – early release to selected "real" users who are expected to report bugs and observations to the developers.
    - Beta – following alpha testing and any necessary further development, release to a wider set of representative users, before final release to all users.

6. Name some **other** forms of testing.
    - Stress testing
    - Regression
    - Usability
    - Security
    - Performance

7. What is **stress testing**? How does it work?
   Stress testing exercises the system beyond its maximum design load.
    - Stressing the system tests failure behaviour
    - Systems should not fail catastrophically
    - Stress testing checks for unacceptable loss of service or data.

   Stressing the system often causes defects to show up. Stress testing is particularly relevant to distributed systems which can exhibit severe degradation as a network becomes overloaded.

   There is also regression testing, which involves re-testing the system following maintenance to ensure continued correctness.

JUNIT

1. What is **JUnit**?
   A framework for programming and managing tests in Java.

2. Describe some of the **features** it provides.
    - Support to conveniently write and manage tests
    - Tools to automatically run the defined tests

3. Describe the process of **writing** and **performing** a JUnit test.
    - Make no change to the Java class to be tested
    - For each class under test, we write a test class (driver)

    - Containing the tests that we want to run
    - Each packaged as a test method (or test case) that calls one or more operations offered by the class under test
    - And checks the results using JUnit asserts
  - JUnit provides Java annotations to tag the parts, e.g. @Test
  - Junit provides test class runners that execute the set of tests in a test class and generate a report
    - Asserts succeed silently
    - Asserts that fail generate a report message and abandon that test method
    - The focus is on highlighting the failures

4. What is a **fixture**? How are they used in JUnit testing?
   Test class may need a common set of objects to be initialized to allow for testing, this is known as a fixture. These are set up before each test method is executed. Use @Before (set up) and @After (tear down).
   Set up -> execute m1 -> tear down -> set up -> execute m2 -> tear down...

5. What **other features** might a test class contain?
   Test classes might also contain global variables, which are useful for fixtures, and private helper methods.

6. What is a **test suite**?
   Test suites are groups of test classes, designed to facilitate easy “one click” regression testing.

7. Describe what is meant by **test-driven development**.
   - Given a specification, write the tests (black box)
   - Then code and keep testing until the tests all succeed
   - The code is ready (or tests are poor)

   Writing tests reduces overall development time.

CONFIGURATION MANAGEMENT

1. Describe what is meant by **configuration management**.
   The ongoing development and improvement of products for many customers, version updates, cross-platform development, etc.

2. Why might there be **parallel versions** of components?
   There may be versions with the same functions but for different OS.

3. Why might there be **sequentially related versions** of components?
   There could be earlier, later, corrected or upgraded versions.

4. What is a **configuration database**? How can it be used?
   A database which stores all the relevant information about configurations; it should be usable to:
   - Assess the impact of changes
   - Generate reports
   - Control system builds for releases
   - Answer questions about clients and their specific release configurations
   - Monitor progress on requested changes

CASE tools are useful here.

5. What is a **CASE tool**?
   Computer Aided Software Engineering tools are used to manage all of the different versions/configurations of the software.

6. What is a **build control tool**?
   A tool used to aid in separate version/configuration creation such as make, Apache Ant and Maven .
7. Name the **three main approaches** and **describe** how they work.
   - make – controlled by files called makefiles, these contain rules specifying how to derive the target program from source files – "declarative"
   - Apache Ant – relatively low level, oriented to Java, uses an XML "build file" to describe the build process; procedural rather than declarative
   - Maven – higher level than Ant, originally for Java but there are plugins for other languages; can be used stand-alone (command line), or as IDE plug-in. Integrated into Eclipse, IntelliJ IDEA, NetBeans

VERSION CONTROL & COLLABORATIVE WORKING

1. What is meant by **version control**?
   Whilst the configuration database contains information about system versions/configurations as selections of components, effectively and instance of the software. At any given time, there would be many more versions of a piece of software than the current release.

2. Understand and describe the **version numbering scheme** i.e. what does X.Y.Z mean?
   - X = Incremented with significant updates i.e. usually backward incompatible, 0 is used only for pre-releases.
   - Y = Incremented with new minor features i.e. backward compatible, resets to 0 after any major change.
   - Z = patch number, incremented when a new build is released with bug fixes that keep backward compatibility. Resets to 0 after any minor change.

3. What is meant by the following terms?
   a. **Branch – the duplication of an object under revision control so that modifications can happen in parallel along both branches**
   b. **Fork – a branch not intended to be merged is usually called a fork**
   c. **Commit – effectively saving the changes made so that they can be pushed back to the main repository**
   d. **Push – merging changes made on the local copy to the main repository version**
   e. **Checkout – taking a copy of file(s) from the repository to work on them**

4. How does **version management** work?

Version management

- VCS tools control a *repository* where the different versions of components are held
  - The versions in the repository are *immutable* – that is once entered *they are never changed*
  - The repository allocates new version numbers when an updated component is deposited
  - … linking back to the original version,
  - … and recording information about the update details
- The VCS can report complete change histories for components, and report the differences between versions
- For effective storage management, component "deltas" are computed and recorded at check-in
  - A delta is a *change* in the component
  - Previous versions can be reconstructed, and might not be held explicitly!

5. What is a **repository**?
   A place where the different versions of components are held.

6. How does a **VCS tool** work?

   - VCS tools usually apply *access control mechanisms* and policies:
     - A programmer must *"check a component version out"* of the repository to work on it
     - … and then must *"check it back in"* when the update is complete (*"commit"* the update)
     - The repository allocates new version numbers at check-in
     - … and usually requires information about the *change* represented by the new version
     - The original version is not discarded
     - The VCS knows and records programmer identities with checked in versions
   - A component might be *locked* whilst checked-out
     - … to prevent an accidental second checkout followed by parallel updates
   - *Or multiple* check-outs might be allowed
     - with *merging* and *conflict resolution* at check-in
   - Also it may be possible to request a *copy* of a version without checking it out
     - … perhaps for testing other components
     - *Copies* may *not* be checked in again
   - Typically administered as a distributed system
     - With the repository on a central server
     - And individual programmers at their own workstations

7. What is a **delta**?
   A change in a component.

8. Name some **well-known** VCS tools.
   - CVS
   - Apache Subversion
   - Git (and GitHub)
   - StarTeam

SOFTWARE ENGINEERING MATHEMATICS & SPECIFICATION

SOFTWARE ENGINEERING SPECIFICATION/MATHEMATICS

1. What is **formal specification**?
   Formal specification means using formal notation to create an abstract model of a system & describe its required properties.

2. Why **use** formal specification?
   Useful in areas of "critical" software where the cost of failure is very high, i.e. spacecraft or medical control systems; they are also used a lot in hardware development.

3. Describe some **advantages** of formal specification.
   - Precision - meaning of formula is unambiguous
   - Conciseness – formulas much more compact than prose
   - Abstraction – can deal with the complexity of large systems by hiding details and focusing on essentials
   - Language independent – mathematical language is not tied to any specific programming language
   - Allows logical analysis – modern formal methods provide automated/interactive analysis of logical properties specification via tools like model-checking

ALLOY OVERVIEW

1. Describe what is meant by the following terms:
   a. **Set – a collection of items**
   b. **Relation – the way in which two or more items are connected**
   c. **First-order logic – uses predicates to tell things about things(?)**

2. What is the **purpose** of Alloy?
   Used to build abstract descriptions of systems and explore their properties.

3. What is the **Alloy Analyser**?
   A feature that provides graphical visualisation of a system and desired properties.

4. Describe what **sig**, **pred** and **run** mean in Alloy.
   - sig – set (signature)
   - pred - predicate
   - run – run

5. What is a **model-based notation**?
   Uses models to represent relations and sets.

BASIC SET THEORY IN ALLOY

1. What is a meant by **univ**, **none** and **int** when referring to sets?
   - univ – the universal set
   - none – the empty set
   - int – set of integers

2. Describe how we can **add our own sets** in Alloy.
   We use for example:
   sig Fruit{} → to introduce a new set
   Or perhaps:
   enum Vegetable {carrot, lettuce, celery}

3. What is a **multiplicity**? Give examples of how they can be used in Alloy, and which ones there are.
   How many members a set contains, for example:
   - one sig Fruit {} → exactly one member
   - lone
   - some

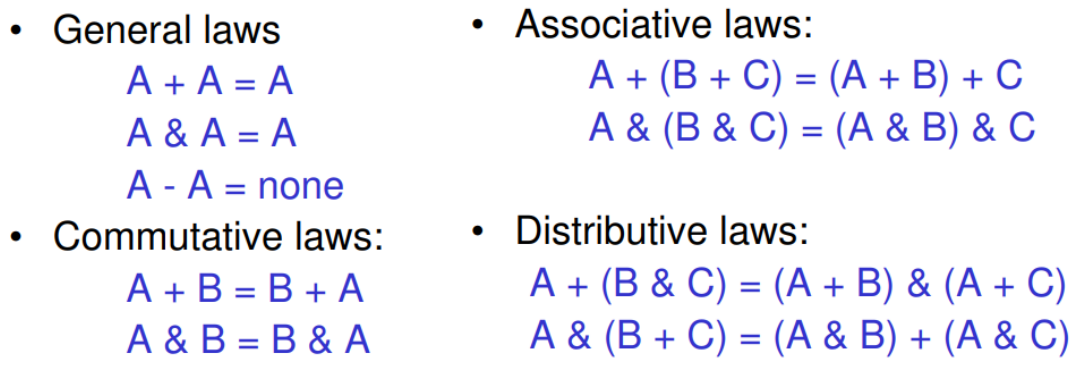
4. How can we create **subsets** of a set in Alloy?
   We can use **extends** for subsets which are mutually distinct, or **in** for ones which may overlap.

5. What is an **abstract signature**?
   Introduces a set that contains nothing apart from the members of sets that extend the signature, such as a boat and train extending vehicle.

6. Describe some of the **operations** we can perform on sets (union, difference, etc.)
   - + - union (in s or in t)
   - & - intersection (in s and t)
   - - - difference (in s but not t)
   - # - number or cardinality (number of members in s)

7. What **logical expressions** can we use with sets? Name and describe some examples.
   - in **subset -**
   - in **membership –** a in s
   - = **equality** – s=t is true if they both have the exact same members
   - some **non-emptiness** – some s is true if s has at least one member
   - no **emptiness** – no p is true if p has no members

8. Give some examples of **general**, **associative**, **commutative** and **distributive laws** in set operations.



9. How might a signature contain **fields**?
   An example of fields within a signature could be:
   sig Aircraft
   {

```
        capacity Int,
        onboard: set Person
}
```

Could think of a signature as a class and a field as an instance variable or attribute.

10. How can a signature be given a **constraint**?
    We could do something like this:

```
sig Aircraft
{
        capacity: Int,
        onboard: set Person
}
{
        # onboard <= capacity      // a constraint
}
```

11. What is a **predicate**? How can we use them in Alloy?
    A parametrised Boolean expression
    For example:

```
pred full [a:Aircraft] {#a.onboard = a.capacity}
run full
```

12. What is a **fact**? How can we use them in Alloy?
    A Boolean expression which imposes some additional constraint on our specification.
    For example:

```
fact manyPeopleExist{ # Person > 2}
```

13. What is an **assertion**? How can we use them in Alloy?
    A Boolean expression expressing some property that we think should follow from our specification.
    For example:

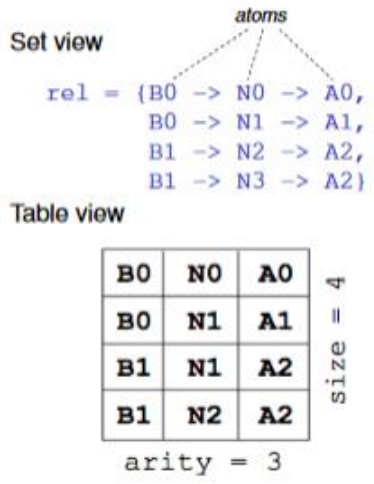
```
assert capacityNonNegative{all a:Aircraft | a.capacity >= 0}
```

And then use **check**

RELATIONS & RELATIONAL OPERATORS

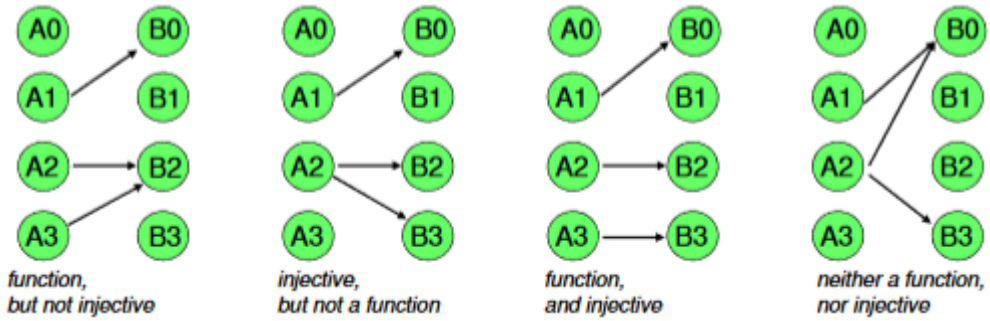
1. What is an **atom**? Describe some properties an atom has.
   An atom is a primitive entity of Alloy which represents an entity whose details are irrelevant. It is indivisible (cannot be broken down into smaller parts), immutable (properties do not change over time) and uninterpreted (does not have any built-in properties).

2. What is a **relation**? How are they **constructed**? Draw an example of them, making sure to specify what **tuple** and **arity** mean.
   A relation is a structure that relates atoms. It consists of a set of tuples (rows); the number of tuples is known as the size, and each tuple is a sequence of atoms that must all have the same length (arity).

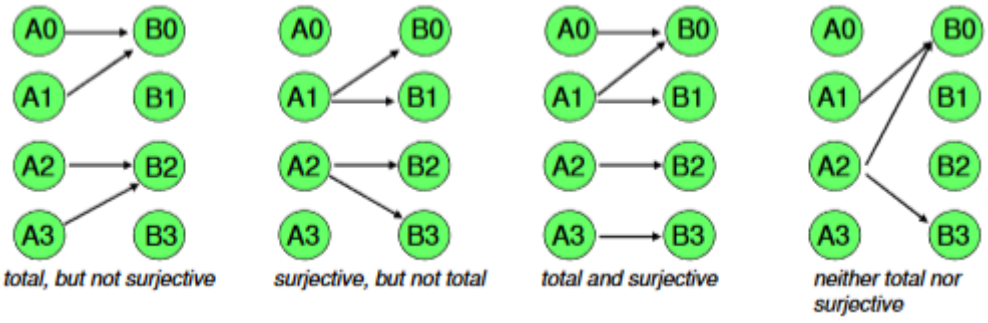


3. What is meant by the following terms when referring to relations:
   a. **Unary – arity of 1 (1 column)**
   b. **Binary – arity of 2**
   c. **Ternary – arity of 3**
   d. **Multirelation – arity of >3**
   e. **Scalar – single values, i.e. myName = {Savi}**

4. Describe the difference between a **function** and **injective relation**. Draw examples of each **combination**.
   - A binary relation that maps each A to at most one B is called a function
   - A binary relation that maps at most one A to each B is injective



5. Describe the difference between a **total** and **subjective relation**. Draw examples of each **combination**.
   - A binary relation mapping each A to at least one B is said to be total
   - A binary relation that maps at least one A to each B is surjective or onto



6. Show how we can represent **multiplicities** in relations.

If a field in a signature has a multiplicity constraint, that determines what kind of relation it is.

```
sig A {
    r1 : B          // r1 maps each A to exactly one B (total and functional - default)
    r2 : one B      // r2 maps each A to exactly one B (total and functional )
    r3 : lone B     // r3 maps each A to at most one B (functional)
    r4 : some B     // r4 maps each A to at least one B (total)
    r5 : set B      // r5 maps each A to any number of B (no constraints)
}
```

A field in a signature may itself be a relation; in this case the field represents a ternary relation. Think of it as a relation mapping each A to a relation from B to C.

```
sig A {
    r1 : B -> C          // no multiplicity constraints
    r2 : B -> some C     // each B maps to at least one C (total)
    r3 : B -> lone C     // each B maps to at most one C (functional)
    r4 : B -> one C      // each B maps to exactly one C (total and functional)
    r5 : B some -> C     // at least one B maps to every C (surjective)
    r6 : B lone -> C     // at most one B maps to each C (injective)
    r7 : B one -> C      // exactly one B maps to each C (surjective and injective)
}
```

7. Describe the difference between **domain** and **range**.
   - The domain of a relation is the set of atoms in its first column
   - The range of a relation is the set of atoms in its last column

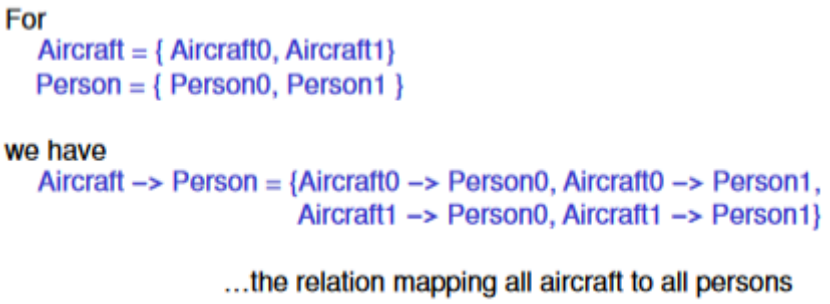
8. What is the **identity relation**?
   The identity relation relates each element to itself.
   For example:
   In a model with the following sets Aircraft and Person
   Aircraft = {Aircraft0, Aircraft1, Aircraft2}
   Person = {Person0, Person1}
   iden has the following value
   iden = {Aircraft0 ->Aircraft0,
   Aircraft1 ->Aircraft1,
   Aircraft2 ->Aircraft2,
   Person0 ->Person0,
   Person1 ->Person1}

MORE ON RELATIONS/RELATIONAL OPERATORS

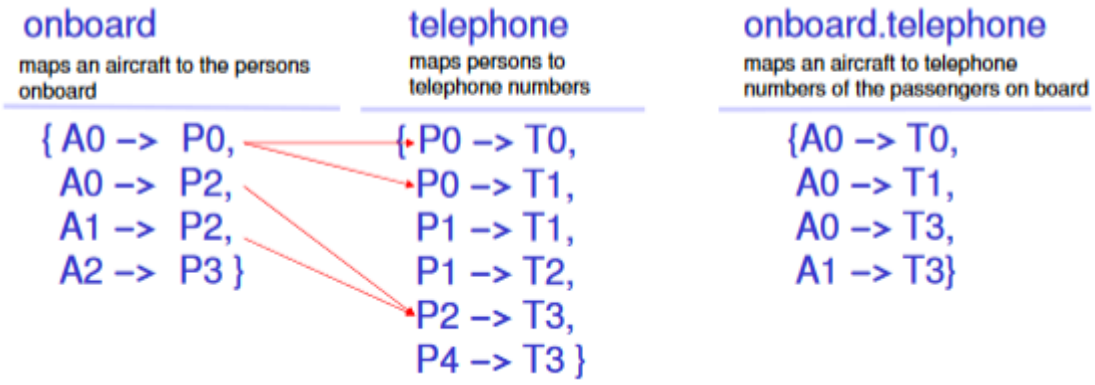
1. Name and draw all the **relational operations** that can be used in Alloy.
   - -> arrow (product) – **p -> q of two relations p and q is the relation consisting of all possible combos of tuples from p and q**
   - . dot (join) – **p.q contains concatenations of tuples from p and q where the values of the last column of p and first column of q agree**
   - [] box (join) – **identical to dot join but binds more tightly (a.b.c[d] is short for d.(a.b.c))**

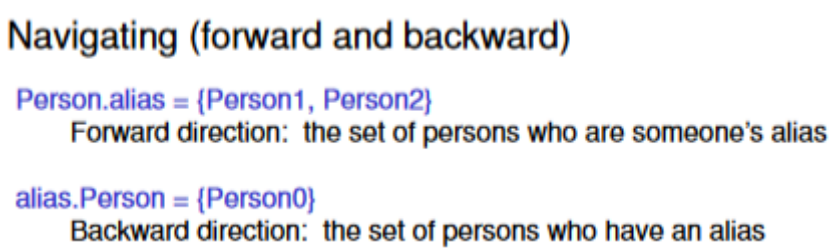
- ^ transitive closure – **binary relation is transitive if, whenever it contains the tuples a->b and b->c, it also contains a->c. The transitive closure ^r of a binary relation r is the smallest relation that contains r and is transitive**
- * reflexive transitive closure – **binary relation is reflexive if it contains the tuple a->a for every atom a in univ. The reflexive transitive closure *r of a binary relation r is the smallest relation that contains r and is both reflexive and transitive**
- ~ transpose (inverse) - **~r of a binary relation r is the relation formed by reversing the order of atoms in each tuple in r**
- <: domain restriction – **s <: r contains those tuples of relation r that start with an element in set s (domain restriction of r to s)**
- :> range restriction – **r :> s contains the tuples of r that end with an element in s (range restriction of r to s)**
- ++ override – **the override p ++ q of relation p by relation q is like the union, except that any tuple in p matching a tuple of q by starting with the same element is dropped. The relations p and q can have any matching arity of two or more.**

2. For all the above relational operations, provide an **example** of how they can be used.
   - -> arrow (product)



   - . dot (join)





   - [] box (join)

Given a relation addr associating names and addresses, the expression

addr[n]

denotes the set of addresses associated with name n, and is equivalent to

n.addr

Box join is very handy when working with ternary relations.

- ^ transitive closure

Example

A relation hasChild which maps each person to their children.

```
hasChild =
{P0 -> P1,
 P0 -> P2,
 P1 -> P3,
 P2 -> P4,
 P4 -> P5}
```

Alternate Intuition: Reachability

The transitive closure of r can also be described as the relation that characterizes the atoms reachable (via the relation) from the each element in the domain of r in **one** or more steps through r.

```
hasChild =
{P0 -> P1,
 P0 -> P2,
 P1 -> P3,
 P2 -> P4,
 P4 -> P5}
```

For example, from P0 we reach...

{P1, P2} in one step
{P3, P4} in two steps
{P5} in three steps

```
^hasChild =
{P0 -> P1,
 P0 -> P2,
 P1 -> P3,
 P2 -> P4,
 P4 -> P5,
 P0 -> P3,
 P0 -> P4,
 P2 -> P5,
 P0 -> P5)}
```

- * reflexive transitive closure

Example

```
Person = {P0, P1, P2, P3}
hasChild = {P0 -> P1,
            P0 -> P2,
            P1 -> P3,
            P2 -> P3}
```

Constants

```
univ = {P0, P1, P2, P3}

iden =
  {P0 -> P0, P1 -> P1, P2 -> P2, P3 -> P3}
```

Transitive & Reflexive Transitive Closures

```
^hasChild = {P0 -> P1, P0 -> P2, P1 -> P3, P2 -> P3, P0 -> P3}
```

...from transitive closure

```
*hasChild = {P0 -> P1, P0 -> P2, P1 -> P3, P2 -> P3, P0 -> P3,
             P0 -> P0, P1 -> P1, P2 -> P2, P3 -> P3}
```

... reflexive tuples

- ~ transpose (inverse)

Given a relation representing an family that maps persons to their children…

```
hasChild = {Person0 -> Person1,
            Person0 -> Person2,
            Person2 -> Person3}
```

…its transpose is the relation that maps persons to their parents…

```
~hasChild = {Person1 -> Person0,
             Person2 -> Person0,
             Person3 -> Person2}
```

- <: domain restriction

Examples

```
hasChild = {P0 -> P1,
            P0 -> P2,
            P3 -> P4}

myself = {P0}
friend = {P4}
```

```
myself <: hasChild =
                {P0 -> P1, P0 -> P2}

hasChild :> friend = {P3 -> P4}
```

- :> range restriction – as above
- ++ override

Example

The capacity of aircraft is increased if the airline squeezes in more seats:

```
capacity = { A0 -> 6, A1 -> 7, A2 -> 5 }
changes = { A0 -> 7), A2 -> 7 }

capacity ++ changes = { A0 -> 7, A1 -> 7, A2 -> 7}
```

3. What is meant by **transitive closure**?
   A binary relation is transitive if, whenever it contains the tuples a->b and b->c, it also contains a->c.
   r.r in r

MATHEMATICAL LOGIC

1. Name and describe all the **mathematical operators**, as well as how they are represented in Alloy.

Verbose	*Shorthand*	*Operator Name*
not	!	negation
and	&&	conjunction
or	||	disjunction
implies	=>	implication
else	,	alternative
iff	<=>	bi-implication

2. Give an **example** of how each operator works.
   - not

p	not p
true	false
false	true

- and

p	q	p and q
true	true	true
true	false	false
false	true	false
false	false	false

- or

p	q	p or q
true	true	true
true	false	true
false	true	true
false	false	false

- implies

p	q	p implies q
true	true	true
true	false	false
false	true	true
false	false	true

- else

The **else** operator is used with the implication operator:

F implies G else H

is equivalent to

(F and G) or ((not F) and H)

...G holds when F holds *...H holds when F does not hold*

- iff

The <=> operator is two-way implication:

p iff q or p <=> q

is equivalent to

(p implies q) and (q implies p)

MORE ON LOGIC

1. List some examples of **equivalent logical expressions**.

p and q	q and p
p and p	p
not not p	p
not (p and q)	(not p) or (not q)
not (p or q)	(not p) and (not q)
p implies q	(not p) or q
p implies q	(not q) implies (not p)

2. What is a **quantified expression**?
   A quantified expression has the form Q x:e | P where Q is a quantifier
   The quantifiers used in Alloy are...

all x:e | P	P holds for every x in e;
some x:e | P	P holds for at least one x in e;
no x:e | P	P holds for no x in e;
lone x:e | P	P holds for at most one x in e;
one x:e | P	P holds for exactly one x in e.

3. Give some examples of **quantified expressions** in Alloy.

```
sig Person
{
  hasChild: set Person
}
```

some p,q: Person| q **in** p.hasChild
...says that there is at least one person who has a child

no p: Person | p **in** p.^hasChild
...says that no person can be reached by following the chain of children from that person (that is there are no cycles in the parent-child relationship)

all p: Person | **some** q: Person | q **in** hasChild.p
...says that every person has at least one parent

MORE ON LOGIC (ELECTRIC BOOGALOO)

1. How can we express **cardinality** in Alloy?

Several quantifier forms can be used to state cardinality constraints on set-valued expressions...

some *s*	*s* contains at least one tuple;
no *s*	*s* contains no tuples (it is empty);
lone *s*	*s* contains at most one tuple;
one *s*	*s* contains exactly one tuple.

Examples

some Person
...says that the set of persons is not empty;

some hasChild
...says that the hasChild relationship is not empty: there is some pair mapping a person to their child (more succinctly than in the example three slides ago)

no (hasChild.Person - Adult)
...says that only adults can be parents

```
sig Adult in Person { }
```

all p: Person | **lone** p.hasChild
...says that every person has at most one child (again, more succinctly than in the example three slides ago);

all p: Person | **one** p.hasChild **or no** p.hasChild
...says the same thing as the expression above.

2. What is a **let expression**? Give an example of how it can be used in Alloy.
   A repeatedly appearing expression, or a subexpression of a larger complicated expression can be factored out with a let expression.

let x = e | A means the same as the expression A with each occurrence of x replaced by the expression e.

Example:

Suppose that, for legal purposes, every person must have at least one heir. If a person has children, they are that person's heirs, otherwise the heir is the queen.

```
sig Person {
  hasChild: set Person,
  heir: some Person
}
one sig queen in Person
```

```
all p: Person |
   let c = p.hasChild |
      (some c  implies p.heir = c else p.heir = queen)
```

This means the same as:

```
all p: Person |
  some p.hasChild
     implies p.heir = p.child
     else p.heir = queen
```

3. What is a **comprehension**? How can we perform them in Alloy?
   Comprehensions form relations from properties – they specify what property a tuple must have for it to belong to a relation.

   Syntax: $\{x_1: e_1, x_2: e_2, \ldots, x_n: e_n \mid F\}$

   Example:
   {p:Adult | no p.hasChild.hasChild} is the set of adult persons who have no grandchildren.

MODEL CHECKING

1. What is meant by **first order logic**?
   A collection of formal systems.

2. What is **model checking**?
   Analyses specifications written in FOL.

3. What is the **signature** of a **specification**?
   The undefined elements (open variables) of a specification.

4. What is the **structure** of a **signature**?
   An assignment of specific values to each of the open variables in the signature.

5. What is the **model** of a **specification**?
   A structure for that specification which makes all the statements in the specification true.

6. Describe the difference between **consistency** and **validity**.
   - A logical statement is **consistent** if there exists at least one model of that statement (There is **at least one** possible world in which the statement is true)
   - A logical statement is **valid** if every structure is a model of that statement (The statement is true in **all** possible worlds)

MORE ON MODEL CHECKING

1. Describe the two main **model checking commands** in Alloy and how they work.
    - Run a predicate – checks whether the predicate is consistent. The analyser tries to find a model in which the predicate is true
    - Check an assertion – checks whether the assertion is valid. The analyser tries to find a model in which the assertion is false (a counterexample)

2. What is meant by **scope**?
   Scope is the number of atoms from each set that Alloy will check – by default it is 3, meaning it only considers and checks models containing 3 atoms from each set.

3. Describe some of the **limitations** of model checking in Alloy.
    - When a predicate is run, if a model instance is found, we can be certain that the predicate is consistent
        - But if no instance found, we cannot conclude that the predicate is inconsistent
        - Might find a model instance by using a larger scope
    - Similarly, when an assertion is checked, if a counterexample is found we can be certain that the assertion is invalid
        - But if no counterexample found, we cannot be sure that the assertion is valid
        - Might find a counterexample by using a larger scope

4. What is meant by the "**small scope**" hypothesis?
   "Most bugs have small counterexamples".

DYNAMIC SYSTEMS

1. Why is it difficult to model **dynamic systems** in Alloy?
   Alloy has no built in notion of state, and variables are mathematical variables, not programming ones; hence no way to assign a new value to a variable.

2. How can we get **around** this?
   Model a change of state by using two variables, representing the before and after states.

3. Describe an example of a **dynamic system**, and how it would be **represented** in Alloy.
   Phone book example:

```
module PhoneBook
sig Name { } // set of names
sig Phone { } // set of phone numbers
sig PhoneBook
{
known: set Name, // names of known persons
number:known -> set Phone // relation mapping known persons to phone number(s)
}
pred show { }
run show

pred PhoneBookExample [pb: PhoneBook]
{
# pb.known > 2 // there are more than two known names
# pb.number > 3 // the number relation contains more than three pairs
# pb.number.Phone >= 2 // the domain of number contains at least two names
some Name - pb.known // there is a name that is not in the known set
some Phone - Name.(pb.number) // there is a phone number that is not in the
// range of the number relation
}
```

```
run PhoneBookExample for 5 but 1 PhoneBook

pred lookupName[
pb : PhoneBook,
n: Name,
result: set Phone
]
{
n in pb.known
result = pb.number[n]
}
run lookupName

pred addName[
pb, pb' : PhoneBook,
n:Name
]
{
n not in pb.known // precondition: name is previously unknown
pb'.known = pb.known + n // postcondition: new name added
pb'.number = pb.number // frame condition: number relation is unchanged
}
run addName for 5 but 2 PhoneBook


pred addNamePhone[
pb, pb' : PhoneBook,
n:Name,
p:Phone
]
{
n in pb.known // precondition: name n is already known
(n -> p) not in pb.number // precondition: n is not already linked with phone p
pb'.number = pb.number + (n ->p) // postcondition: new name-phone pair added
pb'.known = pb.known // frame condition: known names are unchanged
}
run addNamePhone for 5 but 2 PhoneBook

pred deleteNamePhone[
pb, pb' : PhoneBook,
n:Name,
p:Phone
]
{
n in pb.known  // precondition: name n and phone p already known
p in pb.number[n]
(n -> p) in pb.number  // precondition: name and phone p are already linked
pb'.number = pb.number - (n -> p)  // postcondition: name-phone pair removed
pb'.known = pb.known  // frame condition: name and phone remain, just unlinked
}
run deleteNamePhone for 7 but 2 PhoneBook

assert addNamePhoneWorks {
all pb, pb': PhoneBook,
n: Name,
p: Phone,
result: set Phone |
(addNamePhone [pb,pb',n,p] and lookupName [pb',n,result]) implies p in result
}
check addNamePhoneWorks for 3 but 2 PhoneBook

assert deleteNamePhoneReversesAddNamePhone {
```

```
all pb,pb',pb'': PhoneBook,
n:Name,
p:Phone |
(addNamePhone[pb,pb',n,p] and deleteNamePhone[pb',pb'',n,p] )
implies
(pb.known = pb''.known and pb.number = pb''.number)
}
check deleteNamePhoneReversesAddNamePhone

pred deleteName[
pb, pb' : PhoneBook,
n:Name
]
{
n in pb.known // precondition
pb'.known = pb.known - n // first postcondition
pb'.number = pb'.number - (n -> Phone) // second postcondition
}
run deleteName for 5 but 2 PhoneBook

pred deletePhone[
pb, pb' : PhoneBook,
p:Phone,
n:Name
]
{
pb'.number = (pb.number :> (Phone - p)) // postcondition - all (name,phone) paires ending w/ deleted phone number must be
removed
 }
run deletePhone for 5 but 2 PhoneBook
```

4. What do we need to know to specify the **state** of the system?
    - What info is needed to describe the state?
    - How can we represent info in terms of sets and relations?
    - Are there any properties that the system must always satisfy (constraints/invariants)?

5. What do we need to ask to be able to specify a **static operation**?
    - Does the operation require any inputs?
    - Does it produce any outputs?
    - When is it possible to perform this operation (preconditions)?
    - What is the relationship between the current state, the inputs, and the output (postconditions)?

6. What do we need to ask to be able to specify a **dynamic operation**?
    - Does the operation require any inputs?
    - Does it produce any outputs?
    - When is it possible to perform this operation (preconditions)?
    - How does the operation change the system (postconditions)?
    - Are there parts of the system that are unaffected (frame conditions)?

7. What is a **sanity check**? How can it be **performed**?

There are a number of ways that we can check if the specification makes sense:

- Write and run a predicate (often called show) to see models of the system.
- Run a predicate specifying an operation to see a model of what that operation does.
- Write and run new predicates to see how an operation works in some specific situation.
- Write and check assertions to verify general properties of the system and its operations.

PROJECT MANAGEMENT & QUALITY ASSURANCE

COST & EFFORT ESTIMATION

1. Explain why it is important to be able to estimate the **effort** and **cost** that a software project will involve.
2. Discuss what the **difficulties** are in making such estimates.
3. Describe two methods that have been proposed for **project cost estimation**.
4. How would you go about estimating **cost** and **effort** for the case study projects?

ACTIVITY PLANNING

1. Explain the reasons for producing a **project activity plan**.
2. Discuss what is meant by an **"activity"** and how a project manager might go about identifying the different activities that make up a project.
3. Describe in detail the **general structure of a project activity plan**, the kinds of **diagrams** that it might contain, and the ways in which it might be **analysed**.
4. Identify activities and sketch a possible **activity plan** for the case study projects.

TEAM MANAGEMENT

1. Explain why **project team management** has been described as one of the most challenging aspects of **software project management**.
2. Discuss what **factors** should be considered in **selecting** and **training** staff to make up a new team.
3. Write a list of **guidelines** for project managers to follow when **managing teams**, giving reasons for each guideline.
4. Discuss how you would go about **selecting** and **managing** a team for the case study projects.

RISK MANAGEMENT

1. Explain why it is important to **foresee** and **manage risks** in software projects.
2. Discuss the kinds of **risks** that affect software projects.
3. Describe in detail techniques that project managers can use to **foresee risk**, **measure its impact**, and **monitor** and **mitigate** its **effects**.
4. Discuss how you would apply these **risk management techniques** to the case study projects.

QUALITY ASSURANCE

1. Discuss what is meant by **"quality"** in the context of software.

2. Describe some ways in which quality can be **measured**.
3. Describe **techniques** that have been proposed for software quality assurance.
4. Explain what approach you would use for **quality assurance** in the case study projects.

PROJECT MANAGEMENT TOOLS

1. Explain why **software tools** might be helpful in project management.
   They offer a facility for a project team to manage and organise their resources and files much more easily than they could perhaps do with a physical system, i.e. lots of physical paper files and letters and so on. Making this process easier should help the workflow speed up and thus helps the team work to a stricter schedule or get tasks done much faster than normal. Etc.
2. Describe in general the kinds of **functions** that a project management tool should provide.
   - **Planning and scheduling** – Assigning tasks to team members, setting priorities/deadlines, calendars, etc.
   - **Collaboration** – File sharing, communication with team, dashboards
   - **Documentation** – Data accessible from one place, quick access to files
   - **Reporting** – Ability to create customised reports based on project data (budget, expenses, completed tasks, etc.)
   - **Resource Management** – Ability to calculate or outline resources and the cost of their use
   - **Managing Project Budget** – Budget reports/dashboards, time billing/tracking and automated invoicing
3. Give a detailed description of at least **two specific project management tools**, explaining what functions they provide and discussing their **strengths** and **weaknesses**.
   - **Microsoft Project** – Has planning and dynamic scheduling features, dashboards for viewing statuses and remote access to files from a mobile device or other projects, as well as integration with other Office products – very popular and has a good balance of complexity and usability, but can be very expensive if buying for personal or small team use
   - **Zoho Projects** – Ideal for projects with a strict budget, features charts, reports, time/workflow tracking, milestones, calendars and timesheets – cloud-based but very expensive, the cheapest is £120 per year
4. State whether you would recommend one of these **tools** for use in the case study projects, explaining your reasons.