Cheatography

SENG 301 Cheat Sheet

by brownie5 via cheatography.com/27396/cs/7936/

Intro

What is software engineering?

Engineer = capable of designing a system; Programmer = hired to produce code; Developer = design and architect software + documents; Software engineer = thinks of the end product, bridges the gap between customers and programmers

Requirements Engineering

Domain Analysis; Problem Definition; Requirements Gathering; Requirements Analysis; Requirements Specification

Requirements Engineering Details

Design = User Interface Design, Define Subsystems; Modeling = Use Cases, Structural (Formal) Modelling, Dynamic Behavioural Modelling

Quality Assurance

Review and Inspection; Testing; Deployment; Configuration Management; Process Management = Cost Estimation, Planning

Incremental vs Iterative

Incremental	Iterative
1. The requirements are divided into different builds	1. Does not start with a full specification
2. Needs a clear and complete definition of the whole system from the start	2. Building and improving the product step by step
3. Customers can respond to each build (but a build may not represent the whole system)	3. We can get reliable user feedback
4. Incremental fundamentally means **add onto** and helps you to improve your **process**	4. Good for big projects
	5. Only major reqs. can be defined, details may evolve over time
	6. Iterative fundamentally means **redo** and helps you to improve you **product**

Story Cards

Title = Should be a verb description (Ex. View a product location)
Goal = "As a {type of user}, I want to {perform some action} so that I can {achieve some goal}"

Story Maps

Story Maps add narrative structure to a backlog
-Top Level: Main features. Also know as a project backbone
-Second Level: important tasks or related stories. Also known as a walking skeleton.
-Additional tasks are added to flesh out interactions

Prototyping

Types of Prototypes:
Throwaway = Example is a paper one. Will be used only for evaluation; *Incremental* = created a separate components; *Evolutionary* = refined to become actual product
Prototype Fidelity:
Low = Omit details (Rough, no code, easy to trash) - Paper, Storyboard, Wizard of OZ (evaluation) *High* = Looks like a polished product (looks of product, comment on aesthetics, GUI powerpoint etc are used)
Prototyping can help answer:

Prototyping (cont)

- Crowded UI, Knobs versu slider for controlling volume, Navigation = Transparent or solid menu?

White Box/Black Box Testing

White = AKA glass box, structural; Tester know the source code and can debug at runtime = Developer's perspective
Unit Testing = Do discrete parts of my system work as expected?
- Does an individual method work as expected?
- Necessary calls to other methods should be mocked out where possible
-Always white box
Black = Tester gives inputs and observe outputs (No code, only focus on reqs., interacts with UI only) = User's perspective
Acceptance Testing = Is the system working from the customer's perspective? (AKA. System testing)
- Interacts with system through GUI
- Focused in feature
Usually black box

White = Did we build the system right?
Black = Did we build the right system?

Agile vs TDD

TDD = focused on how code gets written (for work cycles of individuals or small groups of developers exclusively)
Agile = Overall development process (focuses on project management and groups of developers, as opposed to specifically how a given developer writes code)

Polymorphism

A property of OO software by which an abstract operation may be performed in different ways, typically in different classes.

Inheritance

Implicit possession by a subclass of features defined in a subclass. Features include variables and methods

Abstract Classes and Abstract Operations

Abstract Operations	No method for that operation exists in the class
Abstract Class	Cannot have any instances

- A class that has one or more abstract more abstract methods must be declared abstract.
- Any class, except a leaf class, can be declared abstract
- Label with <abstract>

Sturctural Modelling

Generalization: Specializing a superclass into subclasses. Avoid unnecessary generalizations

Dependency: Used for extremely weak relationships between classes. Ex. A class makes use of a library

Aggregation: Represents "part-whole" relationships. The whole side is called the aggregate. Aggregations are read as "is part of"

Composition: Are strong forms of aggregation. If the aggregate is destroyed, then the parts are also destroyed.

Aggregation = An association is an aggregation if: the parts 'are part of' the aggregate. The aggregate 'is composed of' the parts. When something owns or controls the aggregate, then they also own and control the parts

Evolvability
"The ability to be evolved" - to adapt in response to change in its environment, requirement and technologies that may have impact on software structural and/or functional enhancements, while taking architectural integrity into consideration
Potential to respond to the pressure to change with minimal modifications
Ex. Bug fixes, enhancements, refactoring, porting
Complexity inherently increases unless work is done to maintain or reduce it

Design Principle 1: Divide and Conquer
Doing something big is normally harder than breaking things up
Separate people can work on each part.
An individual software engineer can specialize
Easier to understand, individual small components
Parts can be replaced or changed without replacing or changing other parts
Ways of dividing: distributed systems = clients and servers; systems = subsystems; subsystem = one or more packages; package = classes; class = methods

Design Principle 3: Reduce coupling
Occurs when there are interdependencies between one module and another
When interdependencies exist, changes in one place will require changes somewhere else
Network of interdependencies makes it hard to see at a glance how some components work
Coupling implies that if you want to reuse one module, you have import the coupled ones too
Types (high coupling to low): - Content: one module to another - Common: two modules share global data, - External : two modules share data format, protocol, - Control - one module controls the flow of another through the argument it passes, - Stamp: Modules share a data structure but each only use a part of it, - Data: modules share data (through parameter passing), - Message: communication between modules via message passing

Adapter
Context = Building an inheritance hierarchy and want to incorporate it into an existing class; the reused class is also often already part of its own inheritance heirarchy
Motivation = how to obtain the power of polymorphism when reusing a class whose methods have the same function but not the same signature as the other methods in the hierarchy?
Pros = allows you to reuse code that doesn't quite match the method signature you were expecting and you can't modify, decouples clients from internal structure
Cons = changes the interfaces to the functionality you want to use, overuse allows for many redundant classes

Singleton
Intent = ensure a class only has one instance and provides a global point of access to it
Motivation = It's important for some classes to have exactly one instance. We want to use a single log object to keep track of when multiple threads are taking certain actions and it's important that the timing is shown correctly
Pros = Ensures only one instance is created
Cons = Better ways of doing this, usually used wrong and is dangerous (security)

Observer

Context = When an association is created between two classes, the code for the class becomes inseparable, reuse of one class means reuse of the other

Motivation = how do you reduce the interconnection between classes, especially between classes that belong to different modules or subsystems?

Antipatterns = Connect an observer directly to an observable so that both have reference to each other, - Make the observes subclasses of the observable

Pros = Limits the amount of information accessed by different classes, ensures that events are handled

Cons = Many modern programming languages have a better, built in event system

Software Architecture

Process of designing the global organization of a software system including:

- Dividing software into subsystems - Deciding how these will interact - Determining their interfaces: the architecture is the core of the design so all software engineers must understand it. Architecture will often constrain the overall efficiency, reusability and maintainability of the system

Importance: To enable everyone to better understand the system, To allow people to work on individual pieces of the system in isolation, prepare for extension of the system, facilitate reuse and reusability

Architecture in different views:

- Logical breakdown into subsystems, - Interfaces among the subsystems, - Dynamics of the interaction among components at run time, - Data will be shared among subsystems, - Components will exist at run time and the machines or devices on which they will be located, -

Ensuring maintainability and reliability = architectural model is stable

- Stable = means new features can be easily added with only small changes to the architecture

Developing an architectural model

Start by sketching an outline:
- Based on the principal reqs. and use cases
- Determine the main components that will be needed
- Chose among the various architectural patterns
- Suggestion: Have several teams independently develop a first draft of the archi. and merge together the best ideas.

Refine the architecture:
- Identify the mains ways in which the components will interact and the interfaces between them
- Decide how each piece of data and functionality will be distributed among the various components
- Determine if you can re-use an existing framework, if you can build a framework.

***Mature the architecture**
- All UML diagrams = useful for describing aspects of the archi. model

Architecture using UML diagrams particular = Package, subsystem, component, deployment

Model View Controller (MVC)

Intent = an architectural pattern used to help separate the user interface layer from other parts of the system

Motivation = I have a program which interacts with advanced user (through command line) and novice users (through a GUI)

Model = Manages behaviour data, responds to requests about its state (view), responds to state change commands (controller)

View - Manages display of information

Controller = Interprets user input, changes model and view

Pros = Separation of concerns, increased usability, readability, reusability and testability

Cons = none worth mentioning

Refactoring

Improving the design of an already written code

The process of changing a software system while not altering the external behaviour of the code

A disciplined way to clean up code that minimize the introduction of bugs

Code Smell = surface indication that corresponds to a deeper problem in the system (Duplicated code, Feature Envy, Middle Man, Temporary Fields)

Refactoring Techniques = Extract method, Move method, Pull Up method, Remove middle man, Extract Class, Inline = put body in caller's method and remove the self method

Duplicated code (same expression in two methods of same class or subclasses/similar code/does same thing, different algorithm) = Extract method, Pull up method/field

Feature Envy (likes other classes than it's own) = Move method, Extract method

Middle man (delegates task to others) = Remove middle man

Temp. field (empty unless needed) = Extract class

Requirements Activities

Eliciting Requirements; Modeling and Analyzing Requirements; Communicating Requirements; Agreeing Requirements; Evolving Requirements

Elicitation = Surveys, analysing existing documents, brainstorming, model driven techniques, observation or card sorting
Modelling = Data, Enterprise, Behavioural, Domain or non-functional reqs.
Communicating = Effective communication among different stakeholders
Agreeing = Verification and validation, requirement conflicts, requirement risks, stakeholder conflicts
Evolving = Managing change, adding reqs, reqs. scrubbing, fixing errors, managing documentation

Overriding

1. SubClassA inherits a method M from A

2. SubClassA implements method M' such that the signatures of M and M' are indistiguishable

3. M' is said to **override** M

Dependency vs. Association

Dependencies only involve using other classes	Associations involve maintaining references to other classes

Aggregation vs Composition

Aggregate parts continue to exist if the aggregation is destoryed and can be used in multiple aggregations	Composite parts are specific to their composition and will be destroyed with the composition

Sequence vs. Communication

Sequence = good for explicit ordering of interactions (Interaction model for use case = use case make time ordering explicit)	**Communication** = Adding details to class diagrams (validates a class diagram and derives an interaction from a class diagram)
Adds detail to messages (Communication has less space)	

Maintainability vs. Evolvability

Maintainability = actual effort required to locate and fix a fault in the program within its operating environment	Evolvability = potential to respond

Evolvability Charcteristics

Integrity	Capability of the software system to maintain architectural coherence while accommodating changes
Changeability	Capability of the system to enable a specified modification to be implemented.
Portability	Capability of the software system to be transferred from one environment to another
Extensibility	Capability of the software system to enable the implementation of extensions to expand or enhance the system (new capabilities and features) with minimal impact to the existing systems
Testability	Capability of the software system to enable modified software to be validated

Design Patterns

The recurring aspects of designs

Pattern = outline of a reusable solution to a general solution encountered in a particular context

Name = unique name for each pattern to ease communication

Intent = Description of the goal of the pattern

Motivation = short scenario illustrating the context in which the pattern can be used

Structure = Class and/or interaction diagrams graphically illustrating the solution

Consequences = Description of the side-effects and results of the pattern

Concurrent Engineering

Divide and Conquer

Teams work on separate components: Follow their own approach

Main Risk: Components don't integrate properly

Design break each other

Common Agile Practice

Refactoring: Incrementally improving the code	Sustainable Pace: No overtime, people work when rested

No sustainable pace because:
-Teams do not have an option to make their own decision
-Allocating people on multiple projects
-Team's inability to say "No"

Agile vs. Spiral

Agile	Spiral
Iterations are shorter (1 to 4 weeks)	Iterations are longer (4 to 6 months)
Not good for low rates of requirements change (cost of collaboration)	Suitable for large scale development (due to risk analysis)
Is good for low-risk and less critical systems	More emphasis on documentation and process

Both are incremental and iterative

Parts of a Use Case

Name = What is this use case about?	Needs to be descriptive so people can use it. The most important part
Actors = Who is going to use this use case?	Focus on types of people
Postconditions = What is the result of this interaction	Focus on what has been accomplished

Name: Add announcement to a single course
Actor: Instructors, TAs
Postconditions: New announcement is added to main Blackboard page for all users. New announcement is emailed to student users.

Integration Testing

Do various parts of the system work together?
- Do subsystems/classes/methods work as expected with other subsystems/classes/methods in the system?
-Do parts of my system work with external dependencies? (database, web services)
-Usually white box

TDD

Specification and not validation (one view), Is a programming technique (another view), Is a way of managing fear during programming, Enables you to take small steps when writing software

Writing the test beforehand makes developers think from a user's perspective when coding leading to a usable API

Static vs Dynamic Testing

Static	Dynamic	Validation	Verification
Objective = Finding errors in early stages of the development cycle	Objective = Checks the functional behaviour of the system	Check that the software product meets the customer's actual needs	Whether the system is well-engineered? Error free?
Are we building the product right?	Are we building the right product?	Dynamic	Static

Static vs Dynamic Testing (cont)

Activities = Reviews, Walkthroughs, Inspection	Testing - The product meets the user's needs = the product fulfills its intended use
The product is built according to the reqs.	

Checking whether the software is of high quality will not ensure that the system is useful. So **Trust but verify, verify but also validate**.

Testing Practices

Exploratory Testing	Simultaneously learning about the software under test while designing and executing tests
	Uses feedback from the last test to inform the next
Brute Force Testing	Testing using every possible input parameters
Equivalence Classes	Divide possible inputs into equivalence classes based on how the system should react to them
	Input in same equivalence class = same system code trigger
	Only one test per equivalence class
	Testers require knowledge: how system works (internally and in detail), how to create input to trigger all code paths

Exploratory = Is a core testing practice for Agile teams
Brute = impossible as you cannot test the whole system
- There is always a limited time for testing and need to focus on testing inputs that will give us the most return on investment

Race Conditions

A race occurs when two threads are using the same resources and the order of operations is important

Critical races can be prevented by locking data so they cannot be accessed by other threads

Ex. A keyword like *synchronized*

Testing Strategies
- Hard to test critical races
- Use mocking to control the order

Why Object Orientation?

OO is primarily a software programming paradigm

OO systems make use of abstraction in order to help make software less complex

OO systems combine procedural and data abstractions = **organizing procedural abstractions in the context of data abstractions**

OO paradigm is an approach which all computations (abstractions) are performed in the context of objects.

OO analysis = which objects are more important for the users (no programming consideration)

Procedural = The entire system in organized into a set of procedures. One main procedure calls the others. (Performs calculations with simple data)

Data = Idea to group together the pieces of data that describe some entity, so that programmers can manipulate the data as a unit

Instance Variables

Attribute	A simple piece of data used to represent the properties of an object
Association	Represents the relationship between instances of one class and instances of another
Static Variable	A variable whose value is shared by all instances of a class
Method in OO	Procedure, function or routine in other programming paradogms
Methods	Procedural abstractions used to implement the behaviour of a class
Operation	Used to discuss and specify a type of behaviour, independently of any code the implements that behaviour (higher level abstraction)

Interface

Has neither instance variables nor concrete methods.
It is a named list of abstract operations
Every single method declared in an Interface will have to be implemented in the subclass

UML Diagrams

Interaction Diagrams: A set of diagrams to model the dynamic aspects of the system. To visualize how the system runs. Often built from a use case and class diagram to illustrate how a set of objects accomplish the required interactions with an actor.

Sequence Diagrams: An interaction diagram that focuses on the sequence of messages exchanged by a set of objects performing a certain task

Communication Diagrams: Emphasize how objects collaborate to realize an interaction

State Diagrams At any given point in time, the system is in precisely one state and will remain in the state until an event occurs to change state. Is a directed graph, nodes are states, edges are transitions. Have timeouts to automatically change states

Interaction Diagrams Show (Interaction) = the steps of the use case, the steps of a piece of functionality. Composed of instances of classes, actors and messages.
Sequence Diagrams: Can represent conditional logic and loops and show explicit destruction of objects.
Communication Diagram: Annotations of object diagrams. Shows link between objects that communicate

UML Modelling

UML (Unified Modelling Language) = graphical language for modelling OO software. 1980s - 1990s = first OO development processes

Types of UML diagrams = Class, objects, interaction, use case, state, activity, component and deployment

Class and object = describe class + methods, relationship between classes

Interaction = How object interact, how system behaves

Use Case = what users can do, feature are related

State + activity = how system behaves internally

Component + Deployment = how the various components of the system are arranged logically and physically

Main symbols = Classes, Associations, Attributes, Operations, Generalizations (groups classes into inheritance hierarchies)
- Associations can be labelled to make explicit associations (are bi-directional by default, can add an arrow) = many to one, many to many, one to one/one to itself (one to one can sometimes be unnecessary, look carefully!)

Design Principle 2: Increase Cohesion

Subsystem or module has high cohesion when related things are kept together and everything else out

Measures the organization of the system, makes it easier to understand and change

Types = functional, layer, communicational, sequential, procedural, temporal, utility

Functional = code that computes a particular results is kept together (easy read, replaceable and reused)

Procedural = keeps procedures together (does not necessarily provide input to the next)

Functional = updating a database, creating a new file or interaction with a user is not functionally cohesive
Procedural = Each individual should have high cohesion in addition to organizing code in objects

Facade

Intent = to simplify the interface to a complicated subsystem

Motivation = I have several parts of a subsystem that is getting quite complicated and I would like to simplify the process for using the subsystem.

Pros = increases readability and testability, reduces coupling

Cons = if your subsystem changes, your facade will need to be updated as well

Requirements

Problem	General Goals (Scheduling a room for a course)
Requirement	All of the things that a system needs to do!; Things you system should (or should not) do; Features your system must provide; Things your users will expect
Functional Requirements	Inputs the system should accept; Outputs the system should produce; Data the system should store that other systems might use; Computations the system should perform (Not algorithms); Timing and sync. of the above (Not response time but the ordering of events)

Functional Requirements: Could relate to interactions with a person or with another system

Functional vs. Non-functional

Functional	What is the system doing? For example: Should be able to make two slides
Non Functional	How is the system doing a thing? For Ex. A created slide should be displayed in 1 second

Non - Functional

Response Time, Throughput, Resource Usage, Reliability, Availability, Failure Recovery, Maintainability, Modularity, Security, Testability, Learnability, Usability, Price, Extensibility, Reusability

Non functional requirements may be more critical than functional requirements, if these are not met, the system is useless! Usually cannot be implemented in a single module of a program.

Planning

Planning = Process of deciding: What activities will be performed, when activities should be started/completed.

Project Planning = Scope of system as a whole, what order features will be done

Iteration Planning: Which features will be included in the next deliverable?

Parts of Planning

Which features are most valuable or risky?

Which features will make it into the project or iteration?

How much effort will each feature take?

Does a given feature depend on other features?

Based on all this: in which order will features be implemented?

Generally the high priority tasks should be picked first for an iteration

Base the amount of work in an iteration on the velocity of your team

70% tasks in the iteration should be must-haves leaving room for uncertainty (based on worst + average case estimates)

Tracking

Process of determining: when and what tasks got completed.

Tracking + planning = extent to which a project in on schedule/cost can be monitored.

TDD Techniques

Triangulation (Playing Difficult):	Referring to how we're using multiple bearings to pinpoint the implementation towards the proper implementation
Using a test double	As alternative and suitable implementations of an interface or class that we don't want to use in a test
	This is because it's too slow, or not available or depends something not available or is just too difficult to instantiate

Objects

Object = a chunk of structured data in a running software system

Represents anything with which you can associate properties and behaviour

Properties characterize the object - describing it's current state

Behaviour = the way an object acts and reacts to the possible changing of its state

Classes

Class = a software module that represents and defines a set of similar objects.

Object with same properties + behaviour = instances of one class

Class contains all of the code that relates to its objects. This includes data for implementing properties and procedures (AKA methods) for implementing behaviour

Classes (cont)

Instance Variables = Each class declares a list of variables corresponding to data that will be present in each instance

Naming Classes
Noun or noun phrase
Singular
Capitalization Style: Pascal Case
No Space: PartTimeEmployee
Do not use underscore (_)
Neither too general nor too specific (city should be municipality)
Avoid reflecting the internals of the system ("Record, Table, Data, Structure, or Information")

Interface vs Abstract

- In Java, a class only derive from one other class = no multiple inheritance in Java (inherit only one abstract class)
- But a class can implement multiple interfaces.
- Abstract classes = meant for inheritance to form a strong relationship between two classes. Can have some implementation code.
- Interface = no method definition/only method headings

When to use?
Abstract = Inheritance (Gives a base class), having non-public members, to add new methods later on
Interface = The API will not change for a while, similar to multiple inheritance, has all public members

Communication Diagram Details

1. The classes of the two objects have an association between them (same direction = unidirectional)
2. The receiving object is stored in a local variable of the sending method. Object created in the sending method or some computation returns an object (<<local>> or [L])
3. A reference to the receiving object has been received as a parameter of the sending method. (<<parameter>> or [P])
4. The receiving object is global. When reference to an object is obtained using a static method. (<<global>> or [G])
5. The objects communicate over a network. (<<network>>)

API

API = application programming interface

An API is provided by a piece of software - abstracts away the implementation of the software

An API is used by other pieces of software. Two pieces interact via API (API acts a contract between them)

The Process of Design

Design = problem solving process to find and describe a way:
- to implement the system's functional reqs.
- respect the constraints imposed by non-functional reqs. (budget, deadlines..)
- adhere to general principles of good quality

Design Issues = sub problems of the overall design. Each issue has several alternative solutions. The designer makes a design decision to resolve each issue. This involves choosing what he or she consider to be the best option from among the alternatives.
Good design = increasing profit with reduced cost, ensure conformation to the reqs., accelerating development, increasing usability, efficiency, reliability, maintainability and reusability

They use knowledge of the reqs., the design created so far, the tech. available, software design principles and 'best practices' and past experiences.