cs2340_exam2 Cheat Sheet
by akschool via cheatography.com/144347/cs/30976/

Test-Driven Development

Failure vs Fault vs Error		
	Failure	Observable incorrect behavior, ex. a+b vs a*b
	Fault (bug):	Related to the code. Failure IFF fault
	Error	Cause of a fault. Usually human error (conceptual, typo, etc.)
Verification	Testing (test cases), Static Verification (all possible inputs), Inspection/review/walkthrough, Formal proof	
Granularity:	Unit Testing -> Integration Testing -> System testing -> Acceptance testing -> Regression testing	
within org	Developers testing -> Alpha testing	outside org: Beta testing -> Product release
what is tdd	Write tests -> write functional code -> refactor	"Make it Fail, Make it Work, Make it Better"
Why TDD	Provides incremental specification, avoid regression errors	
Structure of tests	Set fixture, invoke, check, cleanup	

Teamwork Considerations

People are most important asset		
Critical factors in people management	Consistency, respect, inclusion	
Factors influencing team working	Group composition, Group cohesiveness, Group communications, Group organization	
	Group compo sition	Task-oriented, self-oriented, interaction-oriented
Hitchhiker:	Take credit for team's work w/o contributing	
Couch potato	Willing to work, but drag their feet	
Absorbing leads to couch potatoes / hitchhikers		
- Mirroring reflects consequences onto hitchhikers		

Sequence Robustness

GUI prototype -> Code	Dynamic	Static
	Use Case Model -> Robustness diagram -> Sequence Diagram	Domain Model -> Class Diagram
Robustness diagrams bridge the "what/how" gap		
Notation		
	Boundary Class	a user interface or API class to external system
	Entity Class	a class from the domain model

Sequence Robustness (cont)

	Controller Class	a class representing business logic or logical software function
Valid relationships	Nouns<->Verbs, Verbs<->Verbs	Nouns!->nouns
	valid ex: Actor->Boundary, Boundary<->Controller, entity->controller	
	invalid ex: actor->controller/entity, boundary->entity, entity<->entity, boundary<->boundary	
Robustness analysis guidelines:		
	Make a boundary object for each screen & name them well	
	Usually not real controller objects, but rather logical software functions	
	Direction of arrows not important	
	Boundary/entity classes -> object instances, controllers -> messages	
Sequence Diagrams		
	SD shows how objects within system interact	SSD shows how actors interact w system

Design Class Diagrams

Domain model shows real-world concepts, DCD shows software entities

Class attributes

Full format	visibility name : type multiplicity = default {property-string}
Visibility marks	+ (public), - (private), # (protected)

Attributes assumed private if no visibility is given

Operations assumed public if no visibility is given

Attribution text vs association line

[IMAGE HERE]

Guideline	Use the attribute text notation for data type objects and the association line notation for others

Two ways to show collection attributes

[IMAGE HERE]

Note symbol: can represent UML note or comment, UML constraint, or Method body

Operations and Methods:

Operation syntax, UML1:	visibility name (parameter-list) : return-type = default {property-string}

Operations are usually assumed public if no visibility is shown

Operations to access attributes are often excluded

UML keywords:

Design Class Diagrams (cont)

«actor»:	classifier is an actor, ex: in class diagram, above classifier name
«interface»	classifier is an interface, ex: in class diagram, above classifier name
{abstract}	abstract element; can't be instantiated, ex: in class diagrams, after classifier name or operation name
{ordered}	a set of objects have some imposed order, ex: in class diagrams, at an association end

Dependency:

[IMAGE HERE]

dependency labels are optional	ex: <<call>> and <<create>>

Interfaces, Inheritance, Abstract class, Composition, Aggregation

[IMAGE HERE]

Aggregation	“has-part” association relationship, exists w/o parent
Composition	whole-part association relationship, needs parent to exist

Constraints (3 ways)

[IMAGE HERE]

Utility class

[CODE HERE]

Mapping designs to code

Class-Responsibility--Collaboration (CRC)	Brainstorming tool used in OOD. CRC cards are usually created from index cards.

Mapping designs to code (cont)

CRUFT	useless, redundant, or poorly written code
Don’t Repeat Yourself (DRY)	Every piece of knowledge must have a single, unambiguous, authoritative representation within a system

Separation of concerns (SOC)

	Design principle for separating a computer
	Concern is a set of information that affects the code of a computer program

You Aren’t Gonna Need It (YAGNI)

	A programmer should not add functionality until deemed necessary
	"do the simplest thing that could possibly work”
	Must be used in combination with several other practices, such as continuous refactoring, unit testing and continuous integration

Mapping designs to code (cont)

Collection Classes:	One-to-many relationships are common.	E.g., a Sale must maintain visibility to a group of many SalesLineItem instances

Object visibility

Visibility	the ability of one object to see or have reference to another	
Attribute visibility: B is an attribute of A		
	Relatively permanent visibility	
	Common form of visibility in OO systems	
Parameter visibility: B is a parameter of a method in A		
	Relatively temporary visibility	
	Common to transform parameter visibility into attribute visibility	
Local visibility: B is a (non-parameter) local object in a method of A		
	Relatively temporary visibility	
	Two methods:	- Create a new local instance and assign it to a local variable.
		- Assign the returning object from a method invocation to a local variable.
Global visibility: B is globally visible		

Object visibility (cont)

Preferred method to achieve global visibility is to use the Singleton pattern.

Code smells

code smell	quick-to-spot surface indication that something is wrong with your code	
usually found during examining & refactoring		
usually caused by rushed design and a disregard for technical debt		
	technical debt	the amount of work you create when you try to save time upfront
	right way vs fast way	
Types		
	Bloaters	long method, large class, long parameter list (>=3,4), data clumps (ex: RGB always together)
	Object-Orientation Abusers	Switch statements, Refused Bequest (inherit methods but unused or redefined)
	Change Preventers	

Code smells (cont)

	Divergent Change (many changes to single class from copy-paste)	
	Shotgun surgery (many small changes to many classes from too much coupling, too little cohesion)	
Dispensables	Lazy class (doesn't do enough), Data class (only fields + getters/setters), Duplicated code	
Couplers		
	Feature envy	A method that seems more interested in a class other than the one it is in
	Inappropriate intimacy	Classes know too much about each other's private parts (tightly coupled)
	Middle man:	class performs one action delegating work to other class

Responsibility-driven design

responsibility	Obligation to perform a task or know information
Behavior (doing) vs data (knowing)	
Methods vs responsibilities	methods fulfill responsibilities
	Responsibilities are implemented by means of methods that either act alone or collaborate with other methods and objects

GRASP: [spell out]

Who is responsible for creating a new instance of a class?		
Rules: Assign class B to create class A if:		
	B contains or aggregates A	
	B records A	
	B closely uses A	
	B has the initializing data for A (B is an Expert with respect to creating A)	
If >1 option, prefer aggregation		
1. Creator -> Low coupling:		
Guideline 1	A composite object is an excellent candidate to make its parts	
Guideline 2	Look at the class that has the initializing data	
		E.g., a Payment instance must be initialized with the Sale total. Hence, Sale is a candidate creator of Payment
	Guideline 3	In case of complex rules consider delegation of creation to a helper class
2. Information Expert -> Low coupling, high cohesion, reduce feature envy		
	Assign a responsibility to the class that has the information necessary to fulfill the responsibility	
	Many “partial” information experts may collaborate in a task	
3. Low Coupling	Assign responsibilities so that coupling remains as low as possible.	
	High to low:	
		***Content coupling: one class modifies another (branch into middle of routine, modifies code)
		**Common coupling: share common (global) data
**Control coupling: use a method parameter (by passing some kind of flag) to control a different method		
Stamp/Data coupling: passing complex data or structures between modules(& use primitives when possible)		
Uncoupled: no relationship		
*** DO NOT DO THIS!!!		
** TRY HARD NOT TO DO THIS!		
Common forms of coupling:		
TypeX has an attribute that refers to TypeY		
TypeX calls on services of TypeY		
TypeX has a method that refers to TypeY		
TypeX is a subclass of TypeY		
TypeY is an interface and TypeX implements it		
4. Controller		
UI objects should not have responsibility for fulfilling system events		

GRASP: [spell out] (cont)

Delegates work to other objects & coordinate / control the activity

Assign responsibility to a class that:

Represents the overall System (Façade Controller)

Represents a Use Case scenario where the event occurs (<usecase name>Handler, <ucn>Coordinator, <ucn>Session)

5. High Cohesion: Objects should not do many unrelated things

High to low

***Coincidental: unrelated functions

Logical: multiple logic sections

Temporal: related by phases of an operation

Procedural: required ordering of tasks (addIngredients, mix, bake)

Communicational: operates on same data set

Functional: all essential elements for a single function are in same module (takeOff, fly, land)

GRASP: [spell out] (cont)

	*** DO NOT DO THIS UNLESS UNAVOIDABLE!!

Refactoring:

Goal: Keep program readable, understandable, and maintainable

Preserve behavior by using tests	Ex: rename, extract method, move method, replace temp w query

SOLID: [spell out]

S: Single Responsibility Principle

Each class should have a single overriding responsibility (High Cohesion) -> many small classes > one big class

Each class has one reason why it should change

O: Open/Closed Principle

Objects are open for extension but closed for modification

Extension via inheritance, polymorphism

L: Liskov Substitution Principle

Subclasses should be substitutable for their base classes

class that implements an interface must be able to substitute any reference throughout the code that implements the same interface

I: Interface Segregation Principle

Use several small interfaces vs one larger multipurpose one

SOLID: [spell out] (cont)

	Don’t make clients depend on interfaces they don’t use (Athlete -> SwimmingAthlete, JumpingAthlete)

D: Dependency Inversion Principle

	High-level modules should not depend on low-level modules. Both should depend on abstractions.
	Abstractions should not depend on details. Details should depend on abstractions (writeJava; writeJavaScript -> develop() calls writeJava, writeJavaScript)

ISP vs LSP	ISP: parent <-> client	LSP: parent <-> child