Topic 1: Agile Software Development

1. *Identify and explain what you believe are the four most important concepts related to this topic. Include a concrete example for each of your four concepts. Clearly explain how the four concepts are related to the topic and how they are related to each other.*

I believe daily stand-up, user story, product backlog, and test-driven development are the four most important concepts related to this topic.

- Daily Stand-up – is usually a timeboxed (eg. 15min) meeting where the team meets (typically in front of the task board) at the same time every workday to bring others up to date on any information vital towards coordination and cooperation. For instance, each team member briefly provides updates to the team on any contributions and progress made, or obstacles discovered. In addition, any other topics or discussions that start are cut short and discussed after the meeting instead. To continue, it is common practice to actually stand during the meeting because people are more likely to be quick and brief compared to if everyone was sitting down in a chair. AKA daily scrum, huddle, roll-call, and other similar terms. I think this is important because it enables better communication, collaboration, and organization within a team each day, while also being brief in order to not disrupt work.
- User Stories – are vital to the organization and construction of tasks and workflow. It's important to structure tasks and issue in the proper perspective of an end-user using this type of informal and general explanation of a software feature without incorporating much technical language. It also helps to identify and break down tasks into the smallest components. Not to be confused with software system requirements, user stories are key to putting people first, to have end-users be the center of attention, conversation, and discussion when developing the product/software. Personally, I find user stories very useful because a team member knows the "what" and "why" of the task, including its importance and value overall. In other words, I think this is a core component of agile because it provides a user-focused framework for daily tasks, in turn driving collaboration and creativity to create a better overall product.
- Product Backlog (PB) – is the overall list of tasks to be done, such as: new features, changes to old features (including technical debt like: refactoring, bug fixes, and change requests),infrastructure changes, and any other general activities a team may deliver that has a specific goal or outcome. Practically, the team works on this backlog by prioritizing, timeboxing, and then assigning items for each sprint of work. Each of these items should be somewhat cheap and fast to add or remove items from the product backlog since they should be broken down to their smallest components of work, especially if by doing this, it does not result in direct progress to achieving a desired outcome or solution. In other words, the product backlog represents a single authoritative source of "things" the team works on. Just because something is present in the product backlog, it does not guarantee that it will be delivered – it's just an option to deliver a specific outcome rather than making a commitment. Similarly, I believe the backlog is an important part of agile development because the team can organize and visualize their work.

- Test-Driven Development – is a very useful and important approach to development. It was evolutionary to combine the "test-first" mentality where developers write tests before writing production code. By doing so, the code was written to pass tests which were designed based on business scenarios of practical use. In other words, the overall goal is specification rather than validating if the written code works. Personally, I think test-driven development is very helpful when coding because allows oneself to design with a clear goal and intent further than just an immediate solution. In turn, it becomes easier and faster to create code that fulfills tests and can always be refactored later since the develop writes "just enough" code to make the test(s) pass.

How they're related: Since daily stand-up is typically conducted in front of the task board, it relates to the user stories (tasks) and the product backlog. Sequentially, these relate to test-driven development because this program technique is applied when developing, where tests are written first before production code.

2. *Outline the steps and purpose of the four types of meetings in the Scrum agile process. Indicate the responsibilities of the different members of the team during the meeting.*

Scrum roles in general:

- Scrum Master
    - Often considered a "coach" of the team because they help them do their best possible work
    - Facilitates meetings (including enforcing timeboxes and removing impediments)
        - Organizes meetings, deals with roadblocks and challenges, makes sure the team follows the scrum process
    - Works with PO to ensure PB is ready for next sprint
    - Authority over the process, but not the team members
- Development Team
    - Cross-functional and collaborative team of typically 5-7 members
        - No distinct roles (eg. programmer, designer, tester), but each person may have a specialty or preferred work
    - Everyone works together, helps each other, and shares a deep sense of camaraderie
    - Owns the plan for each sprint and anticipates how much can be completed
- Product Owner (PO)
    - Possesses the vision of what to build and conveys it to the team
    - Focuses on business and marketing requirements, prioritizing all work that needs to be done
        - Declares features to be done (or not done) (including prioritization of the feature list and arbitering requirements).
    - Builds and manages the backlog, providing guidance of which features to do next
    - Interacts with stakeholders and team as middle man to ensure everyone understands the PBIs
    - Not to be confused with project manager

    - Motivates the team with goal and vision, does not manage status or progress of the team

- Sprint Planning: Strictly timeboxed meeting to allow time for the team to plan out the next sprint, which varies depending on the team and organization and can be as short as a week or as long as 4 weeks, although shorter sprints are ideal. Product backlog items (PBIs) are selected for the next sprint and the product owner sets the goals for the sprint, where committed PBIs must have a confirmed definition of done (DoD), including 3 main ideas: properly test, refactored, and is potentially shippable/deliverable. The development team adds initial tasks.
- Daily Scrum (Stand-up): Approximately maximum 15 min, where no electronics are present (besides perhaps the virtual task board) and each team member takes turns explaining what they've done since the last scrum, their plans for today, and any obstacles they are currently facing. If there are any other discussions or issues, they are cut off and noted down to confront after the scrum meeting by the scrum master.
- Backlog Refinement/Grooming: This meeting is no larger than a quarter of the sprint and is also a strictly timeboxed amount of time that allows the team to brainstorm for all the features that are needed for the product. Any PBIs created are typically written as user stories, the PO then prioritizes them, and lastly the team estimates effort (eg. story points, weight, etc.) for each of these items. This meeting can also lead to the refinement of existing PBIs, removing and/or modifying them.
- Sprint Review: This meeting occurs at the end of the sprint and requires the whole team (including scrum master) and PO to be present. The team demos the product, especially to highlight any new and/or changed features that were a result of the sprint. After, there is an opportunity for other audience members, such as the client and/or stakeholders viewing the demo, to ask questions. After the PO finishes answering questions, they declare each PBI "done" or "not done" (there is no partly done). Then, the PO closes the completed tasks (i.e. PBIs) and removes them from the PB, while any unfished tasks go back to the PB. These are commonly prioritized first to be completed in the next sprint.
- Sprint Retrospective: This meeting allows the scrum master and the development team to reflect on and discuss the sprint, the process, what worked, and what didn't work. Retrospective is when the team learns how to improve themselves for the future by focussing on continuing good practices and making improvements to bad habits, so it's important to not skip this step of the meeting.

3. *What does the term timebox mean in the context of scrum? What is its purpose and value to the process? How can it be made to work effectively in practice?*

Timeboxing in the context of scrum refers to the allocation of a fixed, maximum amount of time for an activity. The purpose of this is to define and limit how much time is spent on an activity and is very common because it keeps teams focused on accomplishing the task at hand by providing a clear

definition of done (ie. it ends after the dedicated amount of time is reached). For example, when any meetings are held, they should have an enforced timebox so the team can fully dedicate themselves to that meeting and then continue with other work. By limiting the amount of time for a meeting, people will be able to focus fully for the short amount of time without their mind wandering to other ideas and being less productive during the meeting. In addition, since the meeting is limited time, long meetings won't negatively affect other activities, such as another meeting or development.

4. *Summarize the Manifesto for Agile Software Development. What are the tradeoffs proposed in the document?*

Summary: The value of working software, being flexible in order to respond to changes, and individuals properly interacting and collaborating are more important than the process, contract, tools used, and the original plan. In summary, the manifesto encourages adaptive planning, evolutionary development, early delivery, continuous improvement, and flexible responses to change – all of which are important to developing solutions through the collaborative effort of self-organizing, cross-functional teams and their end-users and customers.

Trade-offs proposed in the document:

- Individuals and interactions over processes and tools
- Working software over comprehensive documentation
- Customer collaboration over contract negotiation
- Responding to change over following a plan

5. Create a chart that shows the similarities and differences between Scrum and any three different Agile processes.

Scrum		**Extreme Programming (XP)**		**Kanban**	
Pros	Cons	Pros	Cons	Pros	Cons
More transparency and project visibility	Risk of scope creep	Fast projects	Code overcomes design	Increases flexibility	Outdated board can lead to issues
Increased team accountability	Team requires experience and commitment	Visible and open communication within the team	Location is difficult if customer is far away	Reduces waste	Teams can overcomplicate the board
Easy to accommodate changes	Wrong/Bad scrum masters can	Reduced costs due to smaller feedback loop	Lack of documentation due to	Easy to understand	Lack of timing

	ruin everything		constant changes		
Increased cost savings	Poorly defined tasks can lead to inaccuracies	Strong teamwork and employee commitment/ satisfaction (eg. pair programming (PP))	Stress due to tight deadlines	Improves delivery flow	
				Minimizes cycle time	

6. You are on a scrum development team that is working on a comprehensive information system for a large bakery. The system contains information about products, ingredients, sales, production schedules and recipes. Your product owner has provided the following user story and your team has decided that it is an epic and must be broken up into product backlog items. Create at least three PBIs from this epic. State any assumptions you make.
"As the production manager I want to identify the best selling products with the least cost and time to bake."

I assume that the production manager will be able to take the intersection of any of the following results, or there will be another user story that is able to combine these different ones.

- “As a production manager, I want to identify the products that sell the most.”
- “As a production manager, I want to identify the products that cost the least to make.”
- “As a production manager, I want to identify the products that take the least amount of time to bake.”

Reference List

Atlassian, “User Stories: Examples and Template,” *Atlassian*. [Online]. Available: https://www.atlassian.com/agile/project-management/user-stories. [Accessed: 11-Dec-2020].

D. Huether, “Agile Practices: Simple Cheat Sheet to Sprint Planning Meeting,” *LeadingAgile*, 11-Feb-2019. [Online]. Available: https://www.leadingagile.com/2012/08/simple-cheat-sheet-to-sprint-planning-meeting/. [Accessed: 11-Dec-2020].

“Daily Meeting,” *Agile Alliance*, 09-Jul-2020. [Online]. Available: https://www.agilealliance.org/glossary/daily-meeting. [Accessed: 11-Dec-2020].

E. Panayotova, “What Are the Pros and Cons of Extreme Programming (XP)?,” *Simple Programmer*, 24-Dec-2018. [Online]. Available: https://simpleprogrammer.com/pros-cons-extreme-programming-xp/. [Accessed: 11-Dec-2020].

K. Eby, “What's the Difference? Agile vs Scrum vs Waterfall vs Kanban,” *Smartsheet*, 15-Feb-2017. [Online]. Available: https://www.smartsheet.com/agile-vs-scrum-vs-waterfall-vs-kanban. [Accessed: 11-Dec-2020].

S. Wambler, “Introduction to Test Driven Development (TDD),” *Agile Data*. [Online]. Available: http://agiledata.org/essays/tdd.html. [Accessed: 11-Dec-2020].

“What Are The Three Scrum Roles?,” *Visual Paradigm*. [Online]. Available: https://www.visual-paradigm.com/scrum/what-are-the-three-scrum-roles/. [Accessed: 11-Dec-2020].

“What are Time-boxed Events in Scrum?,” *Visual Paradigm*. [Online]. Available: https://www.visual-paradigm.com/scrum/what-are-scrum-time-boxed-events/. [Accessed: 11-Dec-2020].

“What is a Backlog?,” *Agile Alliance*, 24-Sep-2019. [Online]. Available: https://www.agilealliance.org/glossary/backlog. [Accessed: 11-Dec-2020].

“What is a Sprint Retrospective?,” *Scrum.org*. [Online]. Available: https://www.scrum.org/resources/what-is-a-sprint-retrospective. [Accessed: 11-Dec-2020].

“What is Backlog Grooming?,” *Agile Alliance*, 24-Sep-2019. [Online]. Available: https://www.agilealliance.org/glossary/backlog-grooming/. [Accessed: 11-Dec-2020].

Topic 2: Object-Oriented (OO) Theory

1. *Identify and explain what you believe are the four most important concepts related to this topic. Include a concrete example for each of your four concepts. Clearly explain how the four concepts are related to the topic and how they are related to each other.*

I believe encapsulation, abstraction, generalization/specialization (inheritance), and polymorphism are the four most important concepts related to this topic. I refer to objects instead of classes because I started talking about how real-world objects are modelled in code using "objects" which are instances of classes.

- Encapsulation – is essentially containing partitions of access between parts of the code so that things do not unnecessarily access or cannot access one another. In other words, it provides boundaries and/or wrappers to indicate privacy. For example, this means using keywords like private, protected, public, and static properly for classes. More specifically, an object's "state" can be seen using its private attributes and can only be modified using its public method, which means that other code has to use the public methods to modify the state of the object instead of directly modifying the private attributes of the object. Takeaway – restrict access and only provide it where necessary. Also sometimes described in terms of "information hiding." I believe encapsulation is important in order to ensure OO theory designed code is properly contained and clean. In addition, bad access can lead to more bugs, so it's easier to do restrict things properly and encapsulate in the first place.
- Abstraction: is a way to deal with and visualize the complexity of modelling reality and life in code through programming design. In other words, one can only use details important for the task instead of working with all the details there would be in real life. Using abstraction to create computational models of real-world objects is done through classes, which an instance of can represent an abstracted real-life object. Since separate objects tend to communicate with each other a lot, abstraction helps each object to only expose a high-level mechanism for using it, which should hide internal implementation details and only revel relevant operations. For example, one does not need to know how something works in order to use it – in these cases: they just need to know what they need to do to get a result, not what's going on in the background. In other words, one object just wants the value of another object and doesn't care exactly how that value was provided (i.e. whether is was calculated, retrieved from somewhere else, or simply given). I believe that proper abstraction is very important to OO theory design in order to reduce the information being dealt with and create cleaner code.
- Generalization/Specialization (Inheritance): is very important to OOP since objects are often very similar, but not exactly the same. By efficiently sharing information (data such as attributes) and actions (logic, purpose, etc. via methods), it is possible to create inheritance trees of these related objects. Each time this is done, common parts between objects are moved up into a super(or parent) class and the parts that differentiate them from one another are moved into sub (or child) classes. For example, although birds and mammals are both animals, they are different: eg. birds have wings, fly, have beaks, and lay eggs; mammals breastfeed, have 4 legs, give live birth; but also similar: eg. both are animals, warm-blooded. These commonalities would

be in a super class whereas the differences would be in sub classes if these were to be modelled in code. Note that super classes may also be subclasses in large hierarchy trees of class inheritance. I think properly using inheritance is important when developing using OOP so there is less code duplication and the code can be cleaner.

- Polymorphism: literally means "many shapes" or "many forms", and refers back to the inheritance hierarchy via method overloading and overriding. In other words, the same thing can be used in multiple different ways. For example, both birds and mammals eat; however, one animals eats with a beak, and the other eats with a mouth that first chews food. Therefore, if the animal class that is the superclass of both the bird and mammal class has an eat method, the subclasses would override it to change it appropriately. Perhaps, the bird class may print "the bird picks up a worm with its beak and swallows it," whereas an mammal may print "the mammal takes a bite out of an apple, chomping on it a few times before swallowing." This example was dynamic polymorphism because it was method overriding, whereas static polymorphism refers to method overloading, which is where more, less, or different parameters can be used in a method call (as long as it has the same name and different parameters: number, type, or both) and the return type doesn't matter. I think polymorphism is also important in order to produce cleaner code and prevent code duplication through efficient OO theory-based design.

How they're related: Abstraction can be thought of as a natural extension of encapsulation since it transitions from hiding information through encapsulation in forms of access to hiding information through abstraction in terms of the amount of necessary information. Inheritance uses the previous two concepts in order to more efficiently create the inheritance trees and share information and actions. Lastly, polymorphism relates back to inheritance since, there can be differences even within similarities. Using polymorphism allows for overriding and/or overloading to model this. For example, both birds and mammals eat; however, one animals eats with a beak, and the other eats with a mouth that first chews food.

2. *Compare and contrast a prototype-based approach to OO to a class-based approach. Include similarities and differences. Include a description of how each approach addresses polymorphism.*

Similarities	Differences
An object (instance) is a specific example of one member of a set	There are no classes in prototype-based, just objects In class-based, a class is an abstract data type bundles with data-type specific functions and relies on being able to generalize characteristics of a set of things

	In prototype-based, a prototype itself is an object instance.
One can be used as a template that is then modified to create a new type (where the new type shares properties of the original) (ie. copying/cloning)	
They both deal with the 4 main concepts of encapsulation, abstraction, inheritance, and polymorphism.	

See more similarities and differences included below as part of the description/explanation for the other part of the question.

While class-based approaches polymorphism by doing things at compile time, prototype-based stores the operations in the prototype data structure, which is copied and modified at run-time. A class is still the equivalence of all objects of it the share the same state space and methods, but adding a method to a prototype effectively makes an element of a new equivalence class. Once can think of a class as a blueprint and an object as the product of that blueprint. Prototypal inheritance and polymorphism is affect because prototype-based OO involves working object instances that inherit direct from other objects.

3. *Explain how dynamic dispatch of methods (late binding) facilitates polymorphic behaviour. Describe how it is implemented in one OO language of your choosing.*

It's a mechanism by which a call to an overridden method is resolved at runtime. This is how Java implements runtime polymorphism. When an overridden method is called by a reference, Java determines which version of that method to execute based on the type of object it refers to. In other words, the type of object which it referred to determines the version of the overridden method that will be called.

4. *Object allocation can be stack-based or heap-based. Explain the trade-offs for both object allocation and object release for each option.*

Stack allocation is much faster since all it really does is move the stack pointer. Comparable performance can be seen from heap allocation using memory pools, but it comes with slightly more complexity and can lead to headaches. Stack memory is always "hot," and it is much more likely to be in cache than heap-allocated memory. However, it's important to note that there's not only a performance difference. It can be seen to affect the lifetime of objects: stack is really mean for fast and short allocations compared to heap because there's no management for stack-based memory: either push or pop things on it and can't free a chunk used by the stack unless it's on top. In contrast, heap-based memory can be managed: gets memory chunks using requests to the kernel and can split, merge, reuse, and free them.

5. *Define strong typing and weak typing. Describe the advantages and disadvantages of both. Give an example for each in which it would be the preferred approach.*

Strong Typing: Each type of data is predefined as part of the programming language. The compiler enforces the data typing and use compliance. For example, all variables and constants must have identified data types in the code. An advantage is that it imposes a rigorous set of rules on the programmer and guarantees a consistency of results, in turn leading to less typing errors. On the other hand, a disadvantage is that it prevents the programmer from inventing or using a data type not anticipated by the code/program/language itself and limits "creative" uses of data. Although this can also be a disadvantage, it remains a fact that the developer is limited by the strong typing. In general, this is more "safety" built-in, but this can limit what an experienced and knowledgeable pro can do as a trade-off for more regulation. These stricter rules at compile time do not mean that things cannot go wrong during runtime and usually lead to more errors/exceptions during compilation since weak typing may not even check.

This could be the preferred approach for new programmers because it can help teach them good practices and how to use variables and different data types. It will also help them keep better track of things. For instance, many schools choose to teach Java first, as well as other strongly typed languages such as C that we learned in university (even if things like pointers, especially void, aren't as strongly typed).

Weak Typing: AKA loosely typed. In general, typing refers to how implicitly data is converted between data types, and weak typing has lots of implicit conversion that allows some weird, funky, yet sometimes cool, interesting, and useful things to happen. In strong typing, operations would not be allowed due to differing data types. For instance, in Java you cannot add a string to an integer, but in JavaScript, which is weakly typed (you don't declare variable types like in Java), you can.

This could be the preferred approach when you are a professional and not creating a massive project because it is easier and faster to develop. For instance, many people like to use Python, which is weakly typed because you don't declare variable data types.

Reference List

A. Petkov, “How to explain object-oriented programming concepts to a 6-year-old,” *freeCodeCamp.org*, 15-Jan-2020. [Online]. Available: https://www.freecodecamp.org/news/object-oriented-programming-concepts-21bb035f7260/. [Accessed: 11-Dec-2020].

Ankit_Bisht, “Stack vs Heap Memory Allocation,” *GeeksforGeeks*, 07-Jul-2020. [Online]. Available: https://www.geeksforgeeks.org/stack-vs-heap-memory-allocation/. [Accessed: 11-Dec-2020].

C. Martin, “prototype based vs. class based inheritance,” *Stack Overflow*, 01-Jun-1958. [Online]. Available: https://stackoverflow.com/questions/816071/prototype-based-vs-class-based-inheritance. [Accessed: 11-Dec-2020].

“CS1130. Transition to OO programming. Spring 2012 --self-paced version,” *Strong versus weak typing*, 2012. [Online]. Available: https://www.cs.cornell.edu/courses/cs1130/2012sp/1130selfpaced/module1/module1part4/strongtyping.html. [Accessed: 11-Dec-2020].

E. Elliott, “Master the JavaScript Interview: What's the Difference Between Class & Prototypal Inheritance?,” *Medium*, 31-Oct-2020. [Online]. Available: https://medium.com/javascript-scene/master-the-javascript-interview-what-s-the-difference-between-class-prototypal-inheritance-e4cd0a7562e9. [Accessed: 11-Dec-2020].

J. Robertson, “As a JS Developer, This Is What Keeps Me Up at Night,” *Toptal Engineering Blog*, 22-Nov-2018. [Online]. Available: https://www.toptal.com/javascript/es6-class-chaos-keeps-js-developer-up. [Accessed: 11-Dec-2020].

M. Rouse, “What is strongly-typed programming language? - Definition from WhatIs.com,” *WhatIs.com*, 01-May-2008. [Online]. Available: https://whatis.techtarget.com/definition/strongly-typed. [Accessed: 11-Dec-2020].

“Runtime Polymorphism or Dynamic method dispatch,” *studytonight.com*. [Online]. Available: https://www.studytonight.com/java/dynamic-method-dispatch.php. [Accessed: 11-Dec-2020].

T. Gyllebring and Benoît, “Which is faster: Stack allocation or Heap allocation,” *Stack Overflow*, 01-Nov-1957. [Online]. Available: https://stackoverflow.com/questions/161053/which-is-faster-stack-allocation-or-heap-allocation. [Accessed: 11-Dec-2020].

Topic 3: Code Smells

1. *Identify and explain what you believe are the four most important concepts related to this topic. Include a concrete example for each of your four concepts. Clearly explain how the four concepts are related to the topic and how they are related to each other.*

I believe organizing lists by root cause makes the most sense because it helps to identify how to get rid of them too. I believe the OO-violation, change prevention, extraneous statements, and high-coupling smells are the four most important concepts related to this topic using this line of thinking.

- OO-Violation Smell: When the application of OOP principles is incomplete or incorrect. For example, switch statements can often be replaced with polymorphic calls. Also, optional member variables that sometimes only have a value are bad and violate OO. The worst smells are when a subclass doesn't use the superclass properly or when classes are functionally equivalent. In order to have efficient, clean, and easily managed code, it's important to avoid and remove this code smell.
- Change Prevention Smell: When a change is made, many more changes in other places must also be made. As a result, development becomes much more complicated and expensive. For example, when a change requires more changes due to coupling or code duplication which hinders progress. Moreover, more annoying change prevention smell examples are separate class hierarchies that are developed in parallel (ie. when a class is added to one, it must also be added to another). Lastly, the most annoying to deal with is when a change in functionality results in completely gutting, dissecting, and redoing a class. I believe it's important to avoid this code smell and quickly fix it whenever it's noticed in order to more easily manage code, keep it clean, and remove duplication.
- Extraneous Statements Smell: When something is pointless and unneeded and its absence would only make the code easier to understand, cleaner, and/or more efficient. This is represented by bad practices such as too many comments, duplicate code, (sometimes) data classes, dead code (including commented out code) that hasn't been removed, unused code as speculative "we might need this" code. I believe it's important to stop these bad habits and remove such useless statements in order to create cleaner and more efficient code.
- High-Coupling Smell: When there is excessive coupling between classes or if coupling is replaced by excessive delegation. This is represented by code that uses data from aggregate classes more than its own, classes that only delegate work (exception for controller classes), and classes that use the data of other classes directly (without accessors or mutators). Obviously, the last example is terrible practice of OO since encapsulation and abstraction are clearly violated, whereas the first two points relate more to a class that either does too much by taking data from other aggregate classes or by not doing enough and only delegates work to other classes. Both are bad practice and I believe these terrible code smells should be avoided and removed in order to create a better codebase that applies proper practices, which will make managing the code and making modifications much easier later.

How they're related: As stated previously, each of these 4 concepts are related to one another because they are a code smell type organized by root cause. They each deal with similar things involving theory of design violations, duplicate and/or unnecessary code or other statements, and making it difficult to maintain code long-term.

2. *Explain the difference and the relationship between the terms Antipattern and Code Smell.*

A code smell is a warning sign in the code and is something that should be investigated as they are commonly a result of bad practices and should be fixed (commonly by modification or deletion). However, this may or may not be bad since it is only a hint that something might be wrong, which does not guarantee something is wrong. Whereas, and antipattern is a pattern which will lead to a bad or worse design when followed. Notably, concept is similar but the opposite of the typical pattern, which, when followed, will lead to a good or better design. In other words, an antipattern provides guidance from a problem to a bad solution and is also known as a "bad idea" in simpler terms.

3. *Identify four Code Smells found in your own assignments from any second year programming course (2000 level course). Provide an example from that assignment for each of the four Code Smells and describe how you would avoid that smell if you were writing the assignment now.*

In a parser for CIS*2750, I had code duplication in multiple locations that should have been made more modular. I won't provide an example because the should-be-function is 60 lines long, but the first snippet is just a part of it (as you can see a lot of it had similar extraneous statements and whitespace). This is a change prevention code smell because a change to this would have to be made in many other locations instead of just once (ie. shotgun surgery) if this was a proper function. If I was writing the assignment now, I would be much more careful about copy-pasting functionality. Instead, I would turn the copied code into an appropriate function first, test that it works in place of where the code part was before, and then call that same function in the other locations.

Also in the same project (and same picture), there exists an example of the extraneous statements code smell because there are many comments that are pointless and unneeded, which should be deleted since it is a comments / dead code smell (see next page for commented out code and print statements that should've been deleted). In CIS*2430, I had to create a course planner app in Java. If I was writing the assignment now, I would avoid the code smell by making sure that I immediately delete code instead of comment it out, even checking it over later at least once to ensure that extraneous statements such as these have been removed.

The following is an example of OOP principle violation (well not exactly by technicality) and I think these two classes were functionally equivalent because the only differences between the honours and general degrees were the number of credits required (20 vs 15) and the appending of " (Honours)" or " (General)" onto the end of the degree title. This much of a function difference between classes is negligible since it didn't really use anything from the superclass nor do many new things (ie. refused

bequest, and alternative classes with different interfaces) and could have been implemented in their subclasses instead. However, this would lead to the duplication of a little bit of code even if the info of 20 or 15 credits required or Honours/General as an identifier most likely won't change. In response to this, I will also bring up another code smell present here, which is the high-coupling smell because these two abstract classes act as a middle-man and message-chain (due to print chaining and set degree title chaining). The only things it adds as a subclass is the number of credits required, appending the degree honours/general identifier to the degree title, and the common overridden toString() and equals() methods. This can be seen in the second and third groupings of images (as the differences) and the fourth grouping as the commonalities. This addition of more code smell evidence illustrating poor practices leads me to believe it would've been better to remove these two classes and add the honours/general credit requirement amount to the actual degree classes because we can remove 2+ code smells while introducing a very small duplication, thus making a worthwhile trade-off. If I was writing the assignment now, I would carefully examine the design beforehand to verify that middle-man classes such as these that don't do much and pass off functionality above and/or below in the class hierarchy. In this case, where there is a functional equivalent class and high-coupling, I believe this would have been more likely to be caught later during development or during review after. In order to avoid and prevent these code smells, it's a matter of staying vigilant and keeping an eye out for smelly code like this to make sure it's prevented or fixed if sighted.

```
if( i < strlen(entireFile)-2 && (entireFile[i+2] == SPACE || entireFile[i+2] == HTAB) ) {
        // actually unfold the line
        numLines--; // doesn't count as a line

        entireFile[i] = entireFile[i+1] = entireFile[i+2] = '^';
        numFoldedLines++;

        // get first half
        //temp = realloc(temp, strlen(entireFile)-2);
        strncpy(temp, entireFile, i);
        temp[i] = '\0';
        // get second half
        //ending = realloc(ending, strlen(entireFile)-i);
        strcpy(ending, entireFile+i+3);
        ending[strlen(entireFile+i+3)] = '\0';
        // put back
        strcat(temp, ending);
        strcpy(entireFile, temp);

        /*printf("\n\ttemp @index=%d:\n%s\n", i, temp);
        printf("\n\tending @index=%d:\n%s\n", i, ending);*/
        i--;
```

```
abstract public class HonoursDegree extends Degree implements Serializable
{
    // ATTRIBUTES
    private static final double CREDITS_REQUIRED = 20.00;
```

/

```
abstract public class GeneralDegree extends Degree implements Serializable
{
    // ATTRIBUTES
    private static final double CREDITS_REQUIRED = 15.00;
```

&

```
@Override
public void setDegreeTitle(String title)
{
    super.setDegreeTitle(title + " (Honours)");
}
```

/

```
@Override
public void setDegreeTitle(String title)
{
    super.setDegreeTitle(title + " (General)");
}
```

```
@Override
public String toString()
{
    return super.toString() + "\nTotal Credits Required:\t" + CREDITS_REQUIRED;
}
@Override
public boolean equals(Object p)
{
    if(!(p instanceof GeneralDegree)) {
        return false;
    } else {
        return super.equals((Degree)p);
    }
}
```

```
public GeneralDegree()
{
    super();
}
```

```
public GeneralDegree(String title, ArrayList<Course> reqCourses, CourseCatalog catalog)
{
    super(title, reqCourses, catalog);
}
```

```
@Override
abstract public String getMajor();
```

```
@Override
public double getCreditsRequired() {
    return CREDITS_REQUIRED;
}
```

4. *If you were designing an introductory lecture on Code Smells for a second year course, which two Code Smells would you include? Why would you choose them for introducing Code Smells?*

I would include the two specific code smells of duplicate code, which commonly refers to extraneous/unneeded statements/code and can also represent high-coupling due to duplication (shotgun surgery), and inappropriate intimacy, which refers to poor encapsulation when a class accesses and uses the internal fields and/or methods of another class. I would select these two because I find these to be two of the most common mistakes newer developers make. It leads to worse code and makes modifying and maintaining code difficult besides its effect contributing to bad code design. I feel that it would be very beneficial for newer students to learn earlier in order to identify and avoid these code smells.

5. *If you were designing an introductory lecture on Antipatterns for a second year course, which two Antipatterns would you include? Why would you choose them for introducing Antipatterns?*

I would include the specific antipattern spaghetti code because the poor structure makes it difficult to understand, extend, and optimize code. In turn, this negatively affects the ability to frequently refactor in order to improve and support the software and enable iterative development. I believe new students will find knowing this antipattern very helpful as I have seen quite a lot of spaghetti code that goes all over the place before in this course level. My hope is that the new students are able to identify and avoid poor practices and can head towards a better direction instead, which ideally helps them to learn the importance of easily maintainable and iterable code. In addition, I would include the specific antipattern of cut-and-paste programming because it is also another bad practice that I noticed is very common in this course level. Similarly, I hope students will be able to avoid this once it is brought to their attention because this antipattern can lead to significant maintenance problem and bad codes due to alternative forms of reuse and duplications, which can lead to undesirable change prevention and/or high-coupling issues for code, tests, and/or even documentation.

Reference List

"Design Patterns and Refactoring," *SourceMaking*. [Online]. Available: https://sourcemaking.com/antipatterns/software-development-antipatterns. [Accessed: 11-Dec-2020].

"Design Patterns and Refactoring," *SourceMaking*. [Online]. Available: https://sourcemaking.com/refactoring/smells. [Accessed: 11-Dec-2020].

E. Englund, "Anti-Patterns and Code Smells," *Medium*, 22-Apr-2019. [Online]. Available: https://itnext.io/anti-patterns-and-code-smells-46ba1bbdef6d. [Accessed: 11-Dec-2020].

J. Atwood, "Code Smells," *Coding Horror*, 07-Aug-2019. [Online]. Available: https://blog.codinghorror.com/code-smells/. [Accessed: 11-Dec-2020].

J. W. Mittag, "What is the difference between Code Smells and Anti Patterns?," *Software Engineering Stack Exchange*, 01-Jul-1966. [Online]. Available: https://softwareengineering.stackexchange.com/questions/350085/what-is-the-difference-between-code-smells-and-anti-patterns. [Accessed: 11-Dec-2020].

Topic 4: SOLID Principles of Design

1. *Identify and explain what you believe are the four most important concepts related to this topic. Include a concrete example for each of your four concepts. Clearly explain how the four concepts are related to the topic and how they are related to each other.*

I believe single-responsibility, open-closed, Liskov substitution, and interface segregation principles are the four most important concepts related to this topic.

- **S**ingle-Responsibility Principle (SRP) – states that a class that should only have one reason to change (ie. a class should only have one job). For example, if I have a bunch of shapes and a class to calculate the sum of areas of these shapes it should only have one job: calculate the sum of the areas. It should not also print or display this info because that violates SRP. Instead, another class should handle that functionality, such as a sum calculator outputter class.
- **O**pen-Closed Principle (OCP) – states that objects or entities should be open for extension, but closed for modification. In other words, a class should be easy to extend without modifying. Using the same shape and area calculator example, consider if the area calculator was only able to get the area of circles and squares. When this is extended to also be able to calculate the sum of areas including more shapes, more if statements would be added to the code to check what shape it is in the given collection of shapes. However, this violates the open-closed principle. Instead, we should remove the logic for area calculation to each of the shapes' classes. In which case, instead of calculating and totaling the area of all the given shapes, the area calculator just takes the sum of all the shapes' area() function calls.
- **L**iskov Substitution Principle (LSP) – states: Let q(x) be a property provable about objects of x of type T. Then q(y) should be provable for objects y of type S where S is a subtype of T. In other words, every subclass should be substitutable for their parent class. Using the same continued shape and area calculator example, consider if there is a new volume calculator class that extends the area calculator class but returns an array of output instead of a single value for the overridden sum() function. This would lead to an issue when using the sum calculator outputter class because it was adding the numerical value to a string, but cannot add the array value to the string from the new subclasss (because it's a strongly typed language). This is a violation of Liskov substation principle that can be fixed by changing the new volume calculator subclass to also return a numerical value instead.
- **I**nterface Segregation Principle (ISP) – states that a client should never be forced to implement an interface that it doesn't use or clients shouldn't be forced to depend on methods they don't use. Once again, continuing the same example, consider if a shape interface was being used originally since more shapes were added, and now a volume() function has been added. Of course, only 3D shapes require volume so this forces 2D shapes to implement an interface method that is useless, thus violating ISP. This can be remedied by creating another interface for solid shapes to have volume instead of adding it to the flat shape interface where area is. Then, 3D shapes can implement both interfaces, while 2D interfaces only implement the flat shape interface.

How they're related: As stated previously, each of these 4 concepts are related to one another because they are each a SOLID principle. I have shown using a continual example how each SOLID principle relates to one another.

2. *Describe how a prototype based language can be written to adhere to the SOLID principles. If a principle does not apply, explain why.*

In general, I believe a prototype-based language can be written to adhere to 4 of the 5 SOLID principles because there is one issue that comes to mind that violates LSP (subtypes must be suitable for their base types) how inheritance is dealt with. For example, JavaScript is a prototype-based language that is not statically typed, thus providing freedom to do more dynamic typing. Instead of looking at inheritance, behaviour can be examined. For instance:

```
#js
function move(vehicle){
    if(!vehicle instanceof Vehicle)
        throw Error('not a vehicle');
    // ...
}

function moveDynamic(vehicle){
    if(!vehicle.move)
        throw Error('move function not present');
    // ...
}
```

The second function illustrates that any object that has a move function can be moved, whereas the first function shows that it only works on instances of the constructor function. Since anything can be passed for the formal parameter "vehicle" LSP would be violated in the first function for objects that have a move function and would work in the moveDynamic function but are not some kind of Vehicle since it must be an instance of Vehicle in the first function.

3. *Choose one of the SOLID principles and propose a counter-argument to it that gives a reasoned argument as to why the principle should not be upheld in all cases. Be convincing. Provide examples.*

With regards to SRP, consider the following real-life counter-example: think about the duck vehicle one may most likely see in lakes or rivers as a recreational activity. These are street legal and water-capable, which makes it a great vehicle for a unique and surreal experience that can be on both land and water. However, think about if these are ever being driven on the road. Of course not, that's silly. This is because the duck car-boat can travel on both land and water, but a car drives much better on land and a boat drives much better on water, so most families have both. In general, most families already have a

car so they would get a boat anyways instead of the duck car-boat. This duck vehicle is impractical for normal use even if it fits the niche for a fun and leisurely recreational experience. This counter example shows that even though the duck vehicle is nobody's choice for a regular vehicle because it violates SRP and tries to accomplish both land and water travel, there is still the existing circumstance where it can function as the perfect vehicle.

4. *Assess one of your own assignments from a previous course with respect to the SOLID principles. Describe how you would refactor the assignment to adhere to any principles you did not consider when you wrote the assignment. Give specific examples.*

Referencing an old project where I created a function that uses one of four of the monster's abilities to display info about what happened to the target, there is a clear violation of OCP because if elements/types/attributes were added, removed, or changed, this class would need to be modified too (see all the if statements below). Instead of hardcoding the ability types into this monster class, it should be refactored out into its own class along with the related ability name and description, including any potential status effects it could potentially inflict when hitting a target. This way, for an array of abilities, something like "ability[abilityNum].use(targetWeight)" could be called instead of having all these poorly designed and lazy if statements that barely accomplish the job for the small set of abilities loaded in this.

```
/**
 * action method -- uses a pocket monster's ability and displays info about what happened to the target
 * @param abilityNum - refers to which of the monster's 4 abilities is being used
 * @param targetWeight - the weight of the target (either 'light' or 'heavy')
 */
public void useAbility(int abilityNum, String targetWeight) {
    // declare needed variables
    int rndNum = r.nextInt(16) + 1;
    boolean poisoned = false;
    // display use move and move description
    JOptionPane.showMessageDialog(null, name + " used " + ability[abilityNum][0] + "!",
            "Pocket Monsters Simulation - Training Area", JOptionPane.INFORMATION_MESSAGE);
    JOptionPane.showMessageDialog(null, name + "" + ability[abilityNum][1],
            "Pocket Monsters Simulation - Training Area", JOptionPane.INFORMATION_MESSAGE);
    // calc chance of status condition occuring
    if(type.equals("Water") && abilityNum == 2) {   // 30% burn chance
        rndNum = r.nextInt(10) + 1;
        // Water: Scald
        if(rndNum <= 3) {   //display burn msg
            JOptionPane.showMessageDialog(null, "The target was burned!",
                    "Pocket Monsters Simulation - Training Area", JOptionPane.INFORMATION_MESSAGE);
        } // else nothing // end if
    } else if(type.equals("Fire") && (abilityNum == 2 || abilityNum == 3)) {    // 10% burn chance
        rndNum = r.nextInt(10) + 1;
        // Fire: Flamethrower, Flare Blitz
        if(rndNum == 1) {   //display burn msg
            JOptionPane.showMessageDialog(null, "The target was burned!",
                    "Pocket Monsters Simulation - Training Area", JOptionPane.INFORMATION_MESSAGE);
        } // else nothing // end if
    } else if((type.equals("Electric") && abilityNum == 2) || (type.equals("Normal") && abilityNum == 1)) { // 30% stun chance
        rndNum = r.nextInt(10) + 1;
        // Electric: Thunder; Normal: Body Slam
        if(rndNum <= 3) {   //display stun msg
            JOptionPane.showMessageDialog(null, "The target was paralyzed!\nIt can't move!",
                    "Pocket Monsters Simulation - Training Area", JOptionPane.INFORMATION_MESSAGE);
        } // else nothing // end if
    } else if(type.equals("Electric") && abilityNum > 2) { // 10% stun chance
```

Reference List

C. Schults, “SOLID Principles: Don't Listen to People Who Say You Don't Need Them,” *NDepend*, 12-Feb-2020. [Online]. Available: https://blog.ndepend.com/defense-solid-principles/. [Accessed: 11-Dec-2020].

E. Dietrich, “The SOLID Principles in Real Life - DZone Java,” *dzone.com*, 21-Feb-2018. [Online]. Available: https://dzone.com/articles/the-solid-principles-in-real-life. [Accessed: 11-Dec-2020].

M. Overeem, “Michiel Overeem,” *Inheritance and reuse in a prototype world*, 12-Jan-2012. [Online]. Available: https://www.movereem.nl/inheritance-and-reuse-in-a-prototype-world. [Accessed: 11-Dec-2020].

S. Oloruntoba, “S.O.L.I.D: The First 5 Principles of Object Oriented Design,” *DigitalOcean*, 21-Sep-2020. [Online]. Available: https://www.digitalocean.com/community/conceptual_articles/s-o-l-i-d-the-first-five-principles-of-object-oriented-design. [Accessed: 11-Dec-2020].

Topic 5: DevOps and SQA

1. *Identify and explain what you believe are the four most important concepts related to this topic. Include a concrete example for each of your four concepts. Clearly explain how the four concepts are related to the topic and how they are related to each other.*

I believe encouraging teamwork, accepting feedback, iterating rapidly, and automation are the four most important concepts related to this topic.

- Encouraging Teamwork – is important for DevOps and SQA because they work with many other people and teams. Thus, decreasing interpersonal conflict is vital for DevOps and SQA teams in order to be successful. For instance, being open and using good communication both orally and verbally helps encourage a collaborative work environment where everyone can do their best to engineer solutions. For instance, DevOps helps offload some things from other teams to make sure things are working smoothly for both the product and the developers' regular workflow. I believe that solid teamwork is a vital element of an efficient team.
- Accepting Feedback – is an important concept when considering DevOps and SQA from a developer's perspective because these teams frequently provide feedback about a developer's work in order to improve it. For instance, an SQA member typically will review tests written by a developer that were modelled after practical business scenarios they created. Thus, the SQA person will tell the developer if something is incorrect or could be improved. Similarly, there will be feedback from the DevOps team if a developer is not properly following the workflow process that has been put in place. On the flip side, sometimes developers will have feedback for DevOps and SQA with recommendations or suggestions about how to improve things. Sometimes, these are in the form of complaints that identify areas of pain that the teams should investigate for potential improvements. Therefore, I believe accepting feedback and being open to change is an important concept regarding DevOps and SQA.
- Iterating Rapidly – helps all team members, especially when it's getting closer to deadlines and more pushes are being made to finish things up. In order to smoothly get through this crunch time, being able to iterate and deploy rapidly is essential. For example, when things are due end-of-day Friday, but multiple different tasks are being merged in the afternoon, it's important to be able to deploy and test that all merged functionality is working and doesn't cause any issues.
- Automation – plays a major role in the workflow process and is one of the most important things in my opinion. Not only does it drastically improve speed by removing the manual parts of a process and saving time, it also eliminates possibilities where human error can occur. For example, creating scripts and automating processes saves developers a significant amount of time compared to performing individual pieces manually and also prevents the developer from making mistakes during steps since it is taken care of for them.

How they're related: Consider CI/CD (continuous integration / continuous development/deployment). We can look at this and see that the above 4 topics are related. Automating processes such as this reduces the friction between teams and helps to encourage a more collaborative work environment. In

addition, automating this process will accelerate the feedback loop and allow everyone to receive some immediate feedback from CI/CD. For example, any automated tests and/or event integrations will fail before the new code is merged so developers are able to quickly correct the issues without hindering others' development, in turn shortening the development cycle overall and being able to successfully deploy features faster. To continue, this reliable CI/CD deploy process ensures that code is well-tested and performs consistently and reliably in a production-like environment before it's delivered. As a result, this also helps iterate rapidly. Lastly, the obvious benefit from automating the tests and process through the use of CI/CD can be seen from even the most simple implementations because it requires a robust test suite that can be run against the code for every commit to the main branch.

2. *What's the relationship between the roles of DevOps and Quality Assurance? similarities? differences?*

Essentially, DevOps manages customer and development computing environments, as well as software packages and possibly deployment automation; however, they may not always be involved in security evaluation and architectural decisions concerning these environments. In addition, they may also be a bit more into an IT (information technology) side by having the duty to acquire software licenses and fix employees' laptops. Most importantly, the DevOps jobs is to make sure nothing important is falling over and that everything is working smoothly. On the other hand, the SQA team's job is to test software before release by writing and executing test plans, sometimes also doing the automated testing to some extent since they may write code. If it's a really advanced team, they might even submit patches to developers and/or write code to test other code, especially if they practice white-box testing. Most importantly, the SQA team's role is to ensure a bug-free software (or at least as close to that idea as possible) and they are often seen as the primary advocate for the customer.

Some of the similarities between the two roles include the necessary attention to detail, and the capability to attempt scoping and scheduling for what they plan to deliver, as well as the ability to follow-through on these plans and execute them successfully. Both roles require individuals that are reliable and possess a strong mindset that allows them to stay determined in order to engineer solutions. Additionally, these roles tend to require employees to frequently help others and learn many new things. It's also important that people in both roles are not shy and know when to ask people for more guidance and/or help to properly do their jobs, whether this is for themselves or when they are assisting another.

3. *Explain in some detail how the git versioning system works. Include a description of the purpose and use of branches and of the merge process.*

The git versioning control system (VCS) is a distributed version control system (DVCS) and is represented by one main (master) branch, where each branch is either owned by someone or in a server. Regardless, each branch exists and can be pulled by anyone and allows for multiple people to work on it at the same time. When a developer is working on a branch, where it is common taboo to directly work on master,

they need to first pull the branch, which creates a local copy of the branch on their computer. Once finished, the developer pushes their local branch changes to that branch, and then perhaps merges that branch with another (eg. master or a different branch). If everything is successful, then it all works out; however, one may run into some issues if multiple people are working on the same branch concurrently, including when multiple people merge to a branch (eg. master). For instance, if one developer merges after another and they both touched some overlapping parts of the code, the second developer will need to first fix any merge conflicts and retest to make sure that the other developer's code that they pulled in after they merged works properly along with their own changes before reattempting to push or merge. This built-in merge process is purposed to allow for the simultaneous development in the same location and aims to help facilitate this problem and help avoid and resolve conflicts as much as possible. Through the use of git, a DVCS, rather than simply checking out and locking files, the repository, including its history, is fully mirrored. Consequently, if any single server or computer dies, there should exist another full copy elsewhere that can easily be used to restore it since every clone can also be used as a full backup of all the data.

4. *Discuss the differences between software reliability and software quality. What tools and techniques can software teams use to ensure both?*

While quality reflects how well software complies/conforms to a design based on functional requirements and/or specifications, reliability is represented by the probability of failure-free software operation for a specified period of time in a particular environment.

To improve both software reliability and quality, try to keep the size of the developed product as small as possible, use smaller teams of people, and regularly invest in the team's working environment to improve efficiency and effectiveness.

5. *Should DevOps be part of the SQA department or part of the software development team? Why do you feel that way? Support your opinion with examples or references.*

I think its dependant on what the DevOps team is tasked with, and this varies based on the organization. However, I feel that DevOps typically belongs more on the software development side of things because they develop software and write code in order to make the developers' lives easier and automate processes. For example, the DevOps team at my past workplace dealt with the custom code language created there as well as how the product is deployed in our product environment, in local cluster computers, and for the clients' setup. They developed software in order to use the customer language including a debugger as well as many other features. It was clear that the DevOps team also developed this as a software besides dealing with automation and developers' work processes. However, it still remains true that SQA is a key component in the methodology behind DevOps, but I still think DevOps belongs closer to the software development side due to my previous workplace experience. Perhaps this may change in the future as I experience more for myself.

Reference List

"1.1 Getting Started - About Version Control," *Git*. [Online]. Available: https://git-scm.com/book/en/v2/Getting-Started-About-Version-Control. [Accessed: 11-Dec-2020].

Contributor, "The Role of QA in the DevOps World," *DevOps.com*, 02-Mar-2020. [Online]. Available: https://devops.com/role-qa-devops-world/. [Accessed: 11-Dec-2020].

D. Putnam and C. T. Putnam-Majarian, "Understanding Quality and Reliability," *InfoQ*, 23-Mar-2016. [Online]. Available: https://www.infoq.com/articles/understanding-quality-reliability-qsm/. [Accessed: 11-Dec-2020].

D. Whittle, "An Introduction to DevOps," *DevOps.com*, 02-Apr-2014. [Online]. Available: https://devops.com/introductiontodevops/. [Accessed: 11-Dec-2020].

E. Freeman, "Continuous Integration and Continuous Delivery: Implementing and Benefitting from CI/CD," *dummies*. [Online]. Available: https://www.dummies.com/business/operations-management/continuous-integration-and-continuous-delivery-implementing-and-benefitting-from-ci-cd/. [Accessed: 11-Dec-2020].

E. Freeman, "DevOps For Dummies Cheat Sheet," *dummies*. [Online]. Available: https://www.dummies.com/business/operations-management/devops-for-dummies-cheat-sheet/. [Accessed: 11-Dec-2020].

M. Morgan-Graham, "What's the difference between DevOps and Quality Assurance?," *Quora*, 19-Jan-2015. [Online]. Available: https://www.quora.com/Whats-the-difference-between-DevOps-and-Quality-Assurance. [Accessed: 11-Dec-2020].

Topic 6: Design Patterns

1. *Identify and explain what you believe are the four most important concepts related to this topic. Include a concrete example for each of your four concepts. Clearly explain how the four concepts are related to the topic and how they are related to each other.*

I believe architectural, structural, behavioural, and creational patterns are the four most important concepts related to this topic. I believe that these categorical concepts of design patterns are more important to understand when compared to specific design patterns within a category. If you know the idea behind it, learning a specific design pattern won't be very difficult since it just details how to apply the concepts.

- Architectural Design Patterns – describe broad structural approaches overarching the entire project. I believe it's important to understand this general topic in order to create a more comprehensive understanding of the specific design patterns underlying each category.
- Structural Design Patterns – focus on problems and solutions associated with how classes and objects are organized and integrated to build a larger structure. For example, one option is to "adapt" one interface for a class into one that a client expects, whereas another option is to incorporate elements into a composition with methods for aggregation of children. In other words, these patterns focus on using inheritance and interfaces (ie. class-creation) and defining ways to compose objects in order to obtain new functionality (ie. object).
- Behavioural Design Patterns – address problems associated with the assignment of responsibility between objects and the way communication is affected between objects. For example, one option is where command objects are handled/passed-on to other objects by logic-containing processing objects, whereas another option is where command objects encapsulate an action and its parameters. In other words, these focus on the communication between class' objects.
- Creational Design Patterns – focus on the creation, composition, and representation of objects. For example, one option is to centralize the decision of what factory to instantiate, whereas another option is to centralize the creation of an object of a specific type choosing one of several different implementations. In other words, these patterns focus on class instantiation by using inheritance (ie. class-creation) or delegation (ie. object) effectively to get the job done.

How they're related: As stated previously, each of these 4 concepts are related to one another because they are all categorical concepts of design patterns where the latter 3 fit under the first one. Each of the latter 3 are ways to approach the overarching idea of an architectural design pattern. Furthermore, each of these categorical design pattern concept also has many different specific design patterns that represent ways to practically apply that design pattern concept. What I have shown here is the hierarchical relation of generalization/specialization between the concepts of design patterns.

2. *How does an individual developer take advantage of Design Patterns? What steps should the developer take to apply a pattern to their code?*

In order for an individual developer to take advantage of design patterns, they first need learn about design patterns, including both the prerequisite knowledge and understanding and then the various ways and methods to practically apply it using specific design patterns. For instance, after the individual finishes gathering the prerequisite knowledge, they should first learn more about the topic by examining examples in addition to working with some code for extra hands-on experience. It's difficult to memorize everything, so practicing is very helpful and useful as it allows oneself to more accurately retain the ideas and feelings to implement a design pattern. Additionally, it's much easier to take the time to properly and thoroughly develop a solid foundation and understanding. Once this has been taken care of, the individual should first examine the system/problem well and determine a design pattern to use before starting to actually build it. However, this only works if the individual possess experience to understand the system well-enough prior to building it. Thus, it's more common to instead prototype or build/use something to help understand the problem better first. While conducting these experiments, jot down notes and set up a to-do list where there are potential areas of concern (eg. ugly code, something doesn't feel right, etc.). Lastly, the individual can address these notes and checklist by refactoring it so it adheres to the chosen design pattern after they complete their understanding of the system/problem.

3. *Choose any two online catalogues of software Design Patterns. Write a review that compares the two catalogues using criteria that you select. Make a recommendation about which catalogue to use by the end of your review.*

Let's compare the two different factory method and abstract factory design patterns that are both classified under the creational design pattern category. First, note that factory stands for a function, method, or class that is supposed to be producing something, such as an object, file, or database record, etc. Second, recall that a creation method is basically just a wrapper around a constructor call. Third, note that a static creation method is declared as "static" so it can be called on a class without the need to create an object. Lastly, consider the simple factory method design pattern which describes a class that has one creation method with a large conditional that chooses which class to instantiate and return depending on method actual parameters. Usually, this is represented by a single method in a single class. Now, a factory method design pattern is a creation design pattern that provides an interface for creating objects by allowing subclasses to alter the type of an object that will be created. For instance, think about a creation method in a base class and subclasses that extend it. In any of these cases, using the factory method design pattern is a logical choice. Meanwhile, an abstract factory design pattern is a creational design pattern that allows for the production of families related/dependent objects without specifying their concrete classes. Well, a family of objects would be a set of classes that are all related like a family. For example, a transport + engine + controls set of super classes can have several variant combinations that make a family together, such as car + combustion + steering wheel or plane + jet + yoke. It's important to note that the previously mentioned simple factory method design pattern is not an abstract factory design pattern if it's declared as abstract. Based on the practically of both of these two design patterns, I'd personally recommend using the factory method design pattern because it is more concrete and easier to understand in most cases. However, there may be a case where there are

several permutations/combinations of different classes due to the real-life problem the solution is modelling and I think it's worth examining the effectiveness of the abstract factory method design pattern in this situation. Since there are so many different combinations of transport + engine + controls, it makes sense to use the abstract factory design pattern because the program operates with product families.

4. *Identify two Design Patterns that would have been useful to know about when you were implementing an assignment for any of your second year (2000 level) courses. Explain the assignment briefly, how you feel the design pattern would have applied, and how you would change your implementation now, knowing about those Design Patterns.*

In a past project where I created a student course planner, I would have found the strategy and command design patterns. Both the strategy and command design patterns would've made it much easier to deal with the GUI aspect of all the buttons. Previously, code for each button was accessing logic independently and directly, whereas it would have been more efficient to separate this layer into a new interface or class for the common areas and have them all reference and use that instead. In addition, the command design pattern would allow me to further parameterize method with different requests, delay/queue execution of some requests, and even support some undoable operations and small errors in my program.

5. *Distinguish between the (Simple) Factory, Factory Method and Abstract Factory Design Patterns. Succinctly describe each pattern. Clearly define the similarities and differences of each pattern. Give a concrete example of when each one is the most appropriate to use.*

Let's compare the two different factory method and abstract factory design patterns that are both classified under the creational design pattern category. First, note that factory stands for a function, method, or class that is supposed to be producing something, such as an object, file, or database record, etc. Second, recall that a creation method is basically just a wrapper around a constructor call. Third, note that a static creation method is declared as "static" so it can be called on a class without the need to create an object. Lastly, consider the simple factory method design pattern which describes a class that has one creation method with a large conditional that chooses which class to instantiate and return depending on method actual parameters. Usually, this is represented by a single method in a single class. Now, a factory method design pattern is a creation design pattern that provides an interface for creating objects by allowing subclasses to alter the type of an object that will be created. For instance, think about a creation method in a base class and subclasses that extend it. In any of these cases, using the factory method design pattern is a logical choice. Meanwhile, an abstract factory design pattern is a creational design pattern that allows for the production of families related/dependent objects without specifying their concrete classes. Well, a family of objects would be a set of classes that are all related like a family. For example, a transport + engine + controls set of super classes can have several variant combinations that make a family together, such as car + combustion + steering wheel or plane + jet +

yoke. It's important to note that the previously mentioned simple factory method design pattern is not an abstract factory design pattern if it's declared as abstract. Based on the practically of both of these two design patterns, I'd personally recommend using the factory method design pattern because it is more concrete and easier to understand in most cases. However, there may be a case where there are several permutations/combinations of different classes due to the real-life problem the solution is modelling, and I think it's worth examining the effectiveness of the abstract factory method design pattern in this situation. For example, consider if you have a very broad transport manufacturing system for organization of vehicle parts, etc. Since there are so many different combinations of transport + engine + controls, it makes sense to use the abstract factory design pattern because the program operates with product families. Coming back to the simple factory design pattern, it is really just an intermediate step of introducing either a factory method or abstract factory design pattern. Therefore, it is appropriate to use when you know that you will be using one of the two factory design patterns, but aren't sure yet. Then, this can be introduced instead as a temporary intermediate until the program becomes larger and more experimentation, exploration, and prototyping is done to determine which of the two factory design patterns are more appropriate.

Reference List

"The Catalog of Design Patterns," *Refactoring.Guru*. [Online]. Available: https://refactoring.guru/design-patterns/catalog. [Accessed: 11-Dec-2020].

"Command," *Refactoring.Guru*. [Online]. Available: https://refactoring.guru/design-patterns/command. [Accessed: 11-Dec-2020].

"Design Patterns and Refactoring," *SourceMaking*. [Online]. Available: https://sourcemaking.com/design_patterns. [Accessed: 11-Dec-2020].

"Factory Comparison," *Refactoring.Guru*. [Online]. Available: https://refactoring.guru/design-patterns/factory-comparison. [Accessed: 11-Dec-2020].

"Strategy," *Refactoring.Guru*. [Online]. Available: https://refactoring.guru/design-patterns/strategy. [Accessed: 11-Dec-2020].

Topic 7: Software Architectures

1. *Identify and explain what you believe are the four most important concepts related to this topic. Include a concrete example for each of your four concepts. Clearly explain how the four concepts are related to the topic and how they are related to each other.*

I believe creational, structural, behavioural, and architectural patterns are the four most important concepts related to this topic. Note that there is a difference in granularity between design patterns and architectural patterns of these same category names. I believe that these categorical concepts of software architectural patterns are more important to understand when compared to specific patterns within a category. If you know the idea behind it, learning a specific pattern won't be very difficult since it just details how to apply the concepts. Similar to design patters, there is a certain extent of shared understanding that is transferrable and related knowledge between the two topics.

- Architectural Patterns – describe broad structural approaches overarching the entire project. I believe it's important to understand this general topic in order to create a more comprehensive understanding of the specific design patterns underlying each category. In other words, this level of abstraction will typically relate to patterns that define the overall structure. For example, indicating the relationships among different components/increments or defining the rules for specifying relationships among elements of the architecture, such as packages, components, and subsystems.
- Structural Patterns – focus on problems and solutions associated with things are organized and integrated to build a larger structure from an overall perspective. In other words, these patterns focus on using inheritance and interfaces (ie. class-creation) and defining ways to compose objects in order to obtain new functionality (ie. object) between different elements of the architecture in order to create a strong structure.
- Behavioural Patterns – address problems associated with the assignment of responsibility between objects and the way communication is affected between different components within the architecture. In other words, these focus on the communication between different pieces of the architecture to ensure that components are properly linked with regards to behavioural communication and notifications.
- Creational Patterns – focus on the creation, composition, and representation of the overall architecture. For example, one option is to centralize the decision of what factory to instantiate, whereas another option is to centralize the creation of a specific type choosing one of several different implementations. In other words, these patterns focus on instantiation by using inheritance or delegation when relating components to effectively get the job done.

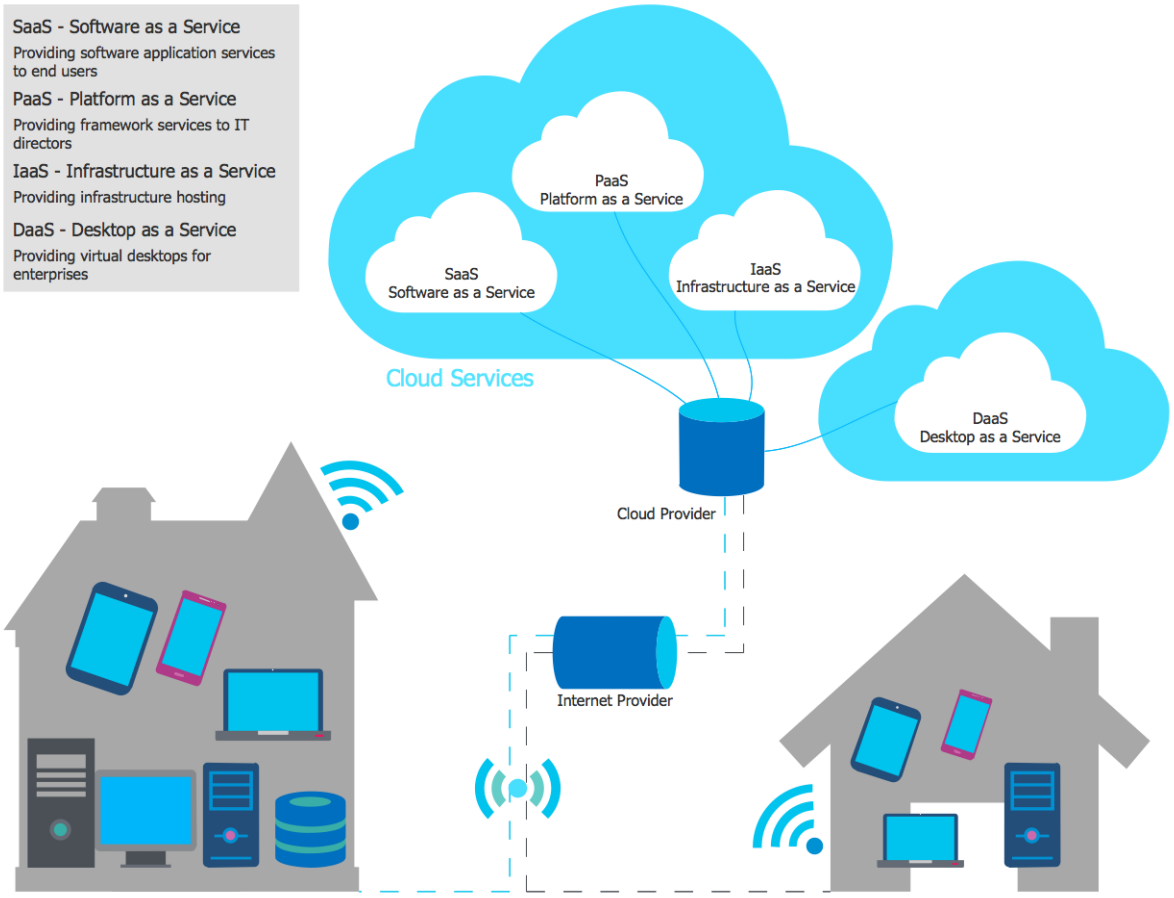
How they're related: As stated previously, each of these 4 concepts are related to one another because they are all categorical concepts of software architecture patterns. Furthermore, each of these categorical pattern concepts also has many different specific patterns that represent ways to practically apply that pattern concept. What I have shown here is the hierarchical relation of generalization/specialization between the concepts of patterns.

2. *What are peer-to-peer systems? Describe how they achieve computation, design considerations, and applications that are well suited to a peer to peer solution.*

A peer-to-peer (P2P) system is where computer systems are connected to one another in a way so that files and can be shared directly between individual computers within the P2P system without the need of a central server. The P2P architecture is designed around several nodes taking part and acting equally as both the client and server. In this way, data is exchanged over TCP IP and the design has an overlay network over TCP IP which enables users to directly connect, in turn taking care of all the heavy lifting complexities. Note that nodes/peers are discoverable and indexed in this overlay network and, based on how these are linked with each other, the network system is classified into structured, unstructured, or hybrid. When designing one of these systems, it's important to consider if you need control over your data: if not, use P2P because it's more open and natural; if yes, you should probably use something like client-server instead that lets you have that. A few examples of applications that are well-suited to a P2P solution are things like Spotify that use a P2P system with their streaming servers to stream multimedia to the users. They are very useful for their quickness and automatic scaling because they are enhanced with more users, so any platform with a shared goal (ie. shared files) can make use of this. The benefit is that there is no server required, so it cuts down the costs a lot, as well as no tensions or worries about traffic surge, servers failing, and other unexpected issues.

3. *Explain the concept of cloud computing for someone who is not very technically inclined. Include at least one sketch or diagram in your explanation.*

Essentially, cloud computing is where everything exists in the "cloud", which is just the internet / at least a server somewhere else, that handles the storage and/or computation of data instead of on your local computer or device. There are many places you utilize cloud computing services. For example, when you save your Google Drive doc, slides, or spreadsheet, it is being saved in a cloud storage system, and this is what allows you to access that same file from any other device as long as you are logged in and have access to it and the internet (of course, provided the host of this file is running properly). Another common example are most streaming services: whether this is for media such as Netflix, music like Spotify, or video games like Google Stadia, it is being streamed from the "cloud" to your computer. In other words, you are able to access this and watch/listen/play even though the files aren't on your device locally because you are accessing it over the internet. Here is a nice diagram to illustrate this:



4. *What is a client-server system? Describe how they achieve computation, design considerations, and applications that are well suited to a client-server solution.*

A client-server system uses the relationship between cooperating programs in an application that is composed of clients initiating requests for services and servers providing that function/service. In a client-server system, a server is in full control. Clients, technically known as thin clients, are just the interface to the data being fetched from the server, which is controlled by an organization, so they have full control and access. Moreover, it's much more difficulty for someone to shut down this network system because they would need to somehow turn off the company's servers. In this relationship, things communicate in a request-response messaging pattern and must adhere to protocols that define the rules, language and dialog patterns used, such as the typical TCP/IP. Therefore, clients send messages to the server to request access to it for something, and the server responds with the function/service/data requested or responds by blocking access. Thus, the client relies on requests to the program in order to access a service on the server. These requests are organized and prioritized in a scheduling system and enables a general-purpose computer to expand its capabilities by utilizing the shared resources of other hosts. When designing this system, it's important to consider if a server is really necessary by asking yourself if you need full control of the data. For example, something with confidential information like a

bank would require a client-server architecture in order to hold full control over data and increase security through restricting access unless the client is verified properly.

5. *Describe the difference between a (thick-) fat-client architecture and a thin-client architecture. Give examples, advantages and disadvantages of each.*

Thin clients are designed to be small so that the bulk of the data processing occurs on the server. It refers to computers without a hard disk drive (HDD) that are designed to serve as clients in client-server architectures and they act as a simple terminal to the server that also requires constant communication with the server. Thin clients provide a "desktop" experience where the end-user has well-defined, regular tasks that use the system. Thin clients are also easy to install and offer a lower total cost of ownership over thick clients. Thick/Fat clients are designed to perform the bulk of the processing in client-server architectures. When using thick clients, continuous server communications are not needed because it is mainly communicating archival storage information to the server. It also refers to the network computer itself, similar to thin clients. A good example that showcases the benefit of thick clients is when applications require multimedia components or large/intensive bandwidth. Simply put, thick clients are good when some operating systems (OSs) or software cannot run on thin clients because thick clients can handle these things since it has its own resources.

Here's a table to easily compare some advantages and disadvantages of each:

Thin Clients		**Thick Clients**	
Pros	Cons	Pros	Cons
Easy to deploy	Needs to validate with the server after data capture	Immediate validation (data verified by client)	More expensive to deploy (and more difficult to set up)
Portability such that all applications are on the server, so any workstation can access	Data collection halts when server is down	Better uptime due to robust technology	Requires more resources
Opportunity to use older/outdated PCs as clients (legacy support)	Cannot be interfaced with other equipment	Only needs intermittent communication with server	Increased security issues
Reduced security threa	Can only run exactly as specified by the server	Requires less servers	
	More downtime	Can store local files and applications	
		Reduced server demand	

Reference List

A. Steed and M. F. Oliveira, “Peer-to-Peer System,” *Peer-to-Peer System - an overview | ScienceDirect Topics*, 2010. [Online]. Available: https://www.sciencedirect.com/topics/computer-science/peer-to-peer-system. [Accessed: 11-Dec-2020].

B, “Designing Effective Peer to Peer Networks,” *Brandon's Blog*, 26-Apr-2020. [Online]. Available: https://skerritt.blog/designing-effective-peer-to-peer-networks/. [Accessed: 11-Dec-2020].

“The Catalog of Design Patterns,” *Refactoring.Guru*. [Online]. Available: https://refactoring.guru/design-patterns/catalog. [Accessed: 11-Dec-2020].

“Command,” *Refactoring.Guru*. [Online]. Available: https://refactoring.guru/design-patterns/command. [Accessed: 11-Dec-2020].

“Design Patterns and Refactoring,” *SourceMaking*. [Online]. Available: https://sourcemaking.com/design_patterns. [Accessed: 11-Dec-2020].

“Factory Comparison,” *Refactoring.Guru*. [Online]. Available: https://refactoring.guru/design-patterns/factory-comparison. [Accessed: 11-Dec-2020].

M. Mouna, “Thin Client Vs Thick Client,” *Medium*, 28-Jun-2018. [Online]. Available: https://medium.com/@mouna.mallipeddi/thin-client-vs-thick-client-69d90c13d02d. [Accessed: 11-Dec-2020].

“P2P,” *P2P (Peer To Peer) Definition*. [Online]. Available: https://techterms.com/definition/p2p. [Accessed: 11-Dec-2020].

S. D. Gantz and D. R. Philpott, “Client Server System,” *Client Server System - an overview | ScienceDirect Topics*, 2013. [Online]. Available: https://www.sciencedirect.com/topics/computer-science/client-server-system. [Accessed: 11-Dec-2020].

Shivang, “P2P Peer to Peer Networks - A Practical Super Helpful Guide To Understanding It,” *8bitmen.com*, 29-Sep-2019. [Online]. Available: https://www.8bitmen.com/p2p-peer-to-peer-networks-oversimplified-everything-you-should-know/. [Accessed: 11-Dec-2020].

“Strategy,” *Refactoring.Guru*. [Online]. Available: https://refactoring.guru/design-patterns/strategy. [Accessed: 11-Dec-2020].

V. Beal, “The Differences Between Thick & Thin Client Hardware,” *Webopedia*, 06-Jul-2006. [Online]. Available: https://www.webopedia.com/DidYouKnow/Hardware_Software/thin_client.asp. [Accessed: 11-Dec-2020].

“What is Client-Server? Definition and FAQs,” *OmniSci*. [Online]. Available: https://www.omnisci.com/technical-glossary/client-server. [Accessed: 11-Dec-2020].

Topic 8: Refactoring

1. Identify and explain what you believe are the four most important concepts related to this topic. Include a concrete example for each of your four concepts. Clearly explain how the four concepts are related to the topic and how they are related to each other.

I believe the composing methods, moving features, organizing data, and refactor by abstraction refactoring technique categories are the four most important concepts related to this topic.

- Composing Methods – is one of the most common (but still important) ways to cleanup dirty code, especially for student projects where methods and functions tend to be too long. The group of refactoring techniques under this category specialize in ways to reduce method length and remove code duplication. I believe writing proper methods is essential to clean code, especially because it limits the chance of code duplication. For example, duplicate code can be turned into a method and then called multiple times instead of being copy-pasted/duplicated (ie. Extract Method refactoring technique).
- Moving Features – is also one of the most common (but still important) ways to cleanup dirty code. This pertains to when features are in a class when they shouldn't be, or it makes more sense to be somewhere else. The group of refactoring techniques under this category specialize in ways to ensure that features are in the correct class. For example, if a field is used more in another class than its own class, that field should be moved to that class and direct everything there instead (ie. Move Field refactoring technique).
- Organizing Data – refers to the data information that flows into or out of methods and how it's organized and stored because this can greatly affect the maintainability of the codebase. Although changes may be as simple as ensuring things are properly encapsulated (eg. data fields are private and can only be accessed through the provided protected/public methods), it is always important to ensure properly encapsulation. Without this, the codebase can quickly get very messy, so properly organizing data is important to writing clean code. For example, make sure that each private attribute/field has appropriate accessor and mutator methods (ie. Self Encapsulate Field refactoring technique).
- Refactor by Abstraction – is AKA dealing with generalization, and this is often the solution to cleaning up messy code. In other words, by moving parts of code to a superclass or subclass one can more appropriately apply OOP for inheritance, polymorphism, and encapsulation, in turn resulting in cleaner code. For example, if two subclasses have the same field, remove the field from subclasses and move it to the superclass (ie. Pull Up Field refactoring technique).

How they're related: As stated previously, each of these 4 concepts are related to one another because they are all categorical refactoring techniques. As you can see from the examples, utilizing each one efficiently will allow a developer to refactor code appropriately in order to create cleaner code that is more efficient, easier to understand, and easier to maintain. Furthermore, each of these categorical refactoring technique concepts also has many different specific refactoring techniques that represent ways to practically apply that refactoring technique idea.

2. *What is the relationship between Code Smells and Refactoring?*

Code smells are indicators of problems that can be addressed during refactoring. These are easy to spot and fix, but they may only be symptoms of a deeper, greater, underlying issue. Also, a code smell doesn't always mean there's something wrong with code, but it at least warrants some investigation to determine that. Usually, code smells can be removed in this case too via refactoring even if there was "nothing wrong." The main goal of refactoring is to deal with technical debt by transforming messy, dirty code into clean code with a better design that is more efficient, easier to understand, and easier to maintain. A good checklist to keep in mind when refactoring: code should be obvious for other programmers, not contain duplication, contain a minimal number of classes and other moving parts, and pass all tests. Clean code that meets all these checklist points is also cheaper to maintain.

3. *List and describe 5 of the main techniques for refactoring code. Choose one of those techniques and provide a before and after code example to illustrate the use of the technique.*

- Composing Methods – is one of the most common (but still important) ways to cleanup dirty code, especially for student projects where methods and functions tend to be too long. The group of refactoring techniques under this category specialize in ways to reduce method length and remove code duplication. I believe writing proper methods is essential to clean code, especially because it limits the chance of code duplication. For example, code segments (especially duplicate code groupings) can be turned into a method and then be called (instead of being copy-pasted/duplicated, etc.) (ie. Extract Method refactoring technique). As exemplified, this technique aims to group a code fragment together and create a new method or function to replace the old code fragment.
    - BEFORE:

```
void printOwing() {
  printBanner();

  // Print details.
  System.out.println("name: " + name);
  System.out.println("amount: " + getOutstanding());
}
```

```
void printOwing() {
  printBanner();
  printDetails(getOutstanding());
}

void printDetails(double outstanding) {
  System.out.println("name: " + name);
  System.out.println("amount: " + outstanding);
}
```

    - AFTER:

- Moving Features – is also one of the most common (but still important) ways to cleanup dirty code. This pertains to when features are in a class when they shouldn't be, or it makes more sense to be somewhere else. The group of refactoring techniques under this category specialize in ways to ensure that features are in the correct class. For example, if a field is used more in another class than its own class, that field should be moved to that class and direct everything there instead (ie. Move Field refactoring technique). As exemplified, this technique aims to move a field to another class when a field is used more in another class than its own class.
- Organizing Data – refers to the data information that flows into or out of methods and how it's organized and stored because this can greatly affect the maintainability of the codebase. Although changes may be as simple as ensuring things are properly encapsulated (eg. data fields are private and can only be accessed through the provided protected/public methods), it is always important to ensure properly encapsulation. Without this, the codebase can quickly get very messy, so properly organizing data is important to writing clean code. For example, make sure that each private attribute/field has appropriate accessor and mutator methods (ie. Self Encapsulate Field refactoring technique). As exemplified, this technique aims to create a getter and setter for a private class field so only it can only be used via those methods instead of directly using them.
- Refactor by Abstraction – is AKA dealing with generalization, and this is often the solution to cleaning up messy code. In other words, by moving parts of code to a superclass or subclass one can more appropriately apply OOP for inheritance, polymorphism, and encapsulation, in turn resulting in cleaner code. For example, if two subclasses have the same field, remove the field from subclasses and move it to the superclass (ie. Pull Up Field refactoring technique). As exemplified, this technique aims to "pull up" a common field from subclasses into the superclass.
- Simplifying Conditionals – refers to when conditionals tend to get increasingly more complicated in their logic over time, and how we can combat and fix this issue. In other words, conditional expressions are often some of the more complicated parts of the code that can be simplified, for example, once all the paths are known. For example, sometimes in large conditionals, a flag of some kind (eg. a Boolean variable) is used, so we can remove this and use break, continue, or return appropriately instead (ie. Remove Control Flag refactoring technique). By doing this, we can at least remove a level of indentation and sometimes simplify the code even more than

that. As exemplified, this technique aims to remove control flags by appropriately using break, continue, and return, especially when they are used for multiple Boolean expressions.

4. *When should refactoring occur in the product development cycle? Support your answer with references and examples.*

A general rule of thumb is to not add functionality at the same time as refactoring. So, if you are blocked when trying to add functionality, first refactor the old code, then implement the new afterwards. Refactoring first makes it easier to add new features too since it's much easier to make changes to cleaner code. Another good general guideline is to only start refactoring the third time you're doing something similar, even if the second time you cringe a bit if you notice (although this isn't the best practice). The most normal time to refactor is during a code review (see https://refactoring.guru/refactoring/when). For example, when there is a merge request, the code is first reviewed and refactored before being merged. This also is true for bug fixes as it eliminates the need for special, technical debt refactoring tasks later.

5. *Choose a catalog of common refactors and review it from the perspective of using the catalog as a developer.*

When looking at the moving features refactoring technique catalog/category, one can see that each specific refactoring technique is to help safely move functionality between classes, create new classes, and hide implementation details from public details (ie. encapsulate properly). The 7 different specific refactoring techniques are all straightforward and one can tell from their name: move field, extract class, inline class, hide delegate, remove middleman, introduce foreign method, and introduce local extension. I agree with the first 5; however, the last 2 are a little suspicious to me as they are very similar and I have trouble fully visualizing how they would be practically applied. To clarify, a feel that the interchangeable solution to the same problem between the two can lead to messy code, the opposite of the goal of refactoring. In my opinion, these techniques are more of a last resort technique if better ways cannot be used. One must be very careful when using these two techniques in order to not introduce bad code when refactoring.

- Introduce Foreign Method – [**problem**] a utility class doesn't contain a needed method and it cannot be added to the class. [**solution**] add the method to a client class and pass an object of the utility class to it as an argument.
- Introduce Local Extension – [**problem**] a utility class doesn't contain some needed methods and they cannot be added to the class. [**solution**] create a new subclass or wrapper class that contains the methods.

Reference List

"Catalog of Refactoring," *Refactoring.Guru*. [Online]. Available: https://refactoring.guru/refactoring/catalog. [Accessed: 11-Dec-2020].

"Extract Method," *Refactoring.Guru*. [Online]. Available: https://refactoring.guru/extract-method. [Accessed: 11-Dec-2020].

"Move Field," *Refactoring.Guru*. [Online]. Available: https://refactoring.guru/move-field. [Accessed: 11-Dec-2020].

"Moving Features between Objects," *Refactoring.Guru*. [Online]. Available: https://refactoring.guru/refactoring/techniques/moving-features-between-objects. [Accessed: 11-Dec-2020].

"Pull Up Field," *Refactoring.Guru*. [Online]. Available: https://refactoring.guru/pull-up-field. [Accessed: 11-Dec-2020].

"Refactoring," *Refactoring.Guru*. [Online]. Available: https://refactoring.guru/refactoring. [Accessed: 11-Dec-2020].

"Self Encapsulate Field," *Refactoring.Guru*. [Online]. Available: https://refactoring.guru/self-encapsulate-field. [Accessed: 11-Dec-2020].

"When to refactor," *Refactoring.Guru*. [Online]. Available: https://refactoring.guru/refactoring/when. [Accessed: 11-Dec-2020].