UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA
Denkleiers • Leading Minds • Dikgopolo tša Dihlalefi

Instructional Design Document

COS750 Exam Assignment
Factory Method

Group: Concrete Creators

Charl Pieter Pretorius: u22519042
Ivan Gareth Horak: u21456552
James Alexander Fawkes Fitzsimons: u21516741

November 15, 2025

Contents

Introduction

Project Information	
Project Title	COS 214 Factory Method App
Project Purpose	Design pattern knowledge supplementation through post-lecture instructional content and assessment.

Roles and Responsibilities	
Project Owner	University of Pretoria, COS 750
Project Stakeholder(s)	Prof. Linda Marshall; COS214 Students

Project Overview

Our project designs and prototypes an app-based learning experience that helps second-year CS students recognise when to apply the Factory Method Pattern (FM) and how to implement it correctly in C++. Using the Dick & Carey model, we translate performance goals into aligned content, practice, and assessments. Instruction blends visuals, narrated explanations and hands-on coding tasks to serve VARK preferences. Gagné's Nine Events structure each lesson from attention-grabbing problem scenarios through guided examples, practice with immediate feedback, and summative checks mapped to Bloom's hierarchy.

ADDIE summarises the path:

- **Where we are going:** Students can recognise when the Factory Method should be used, know how to implement it in C++ and understand and create its UML class diagram.
- **How we get there:** VARK-friendly lessons scripted by Gagné's events, worked examples, and hands-on code practice that targets second-year Bloom levels.
- **How we know we arrived:** formative assessment during the learning stage, diagnostic assessment throughout the whole program to adapt to the current student base and to note lesson weakness for future improvement and finally summative assessment using BLOOM levels to ensure learning outcomes have been sufficiently met.

Note: Should the prototype be implemented it would revise instruction in short review cycles based on diagnostic assessment data, since this is just a prototype it will lack the data to do this.

Furthermore: The ID document is designed from the point of view of teaching the entire factory method pattern; from this it is assumed later in the SE document that most teaching elements as stated in the ID document have already been covered in the in person lectures and it will thusly be focused on the more practical elements and micro questions.

Instructional Design Model (Dick & Carey)

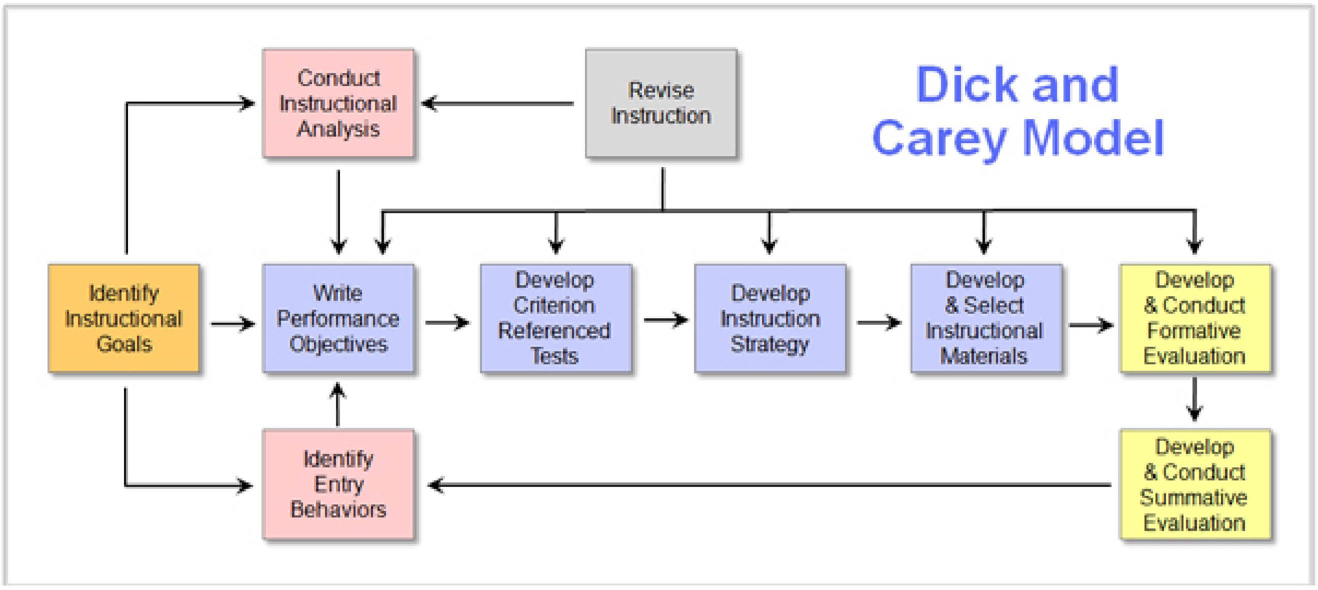
Dick and Carey Model
Conduct Instructional Analysis
Revise Instruction
Identify Instructional Goals
Write Performance Objectives
Develop Criterion Referenced Tests
Develop Instruction Strategy
Develop & Select Instructional Materials
Develop & Conduct Formative Evaluation
Identify Entry Behaviors
Develop & Conduct Summative Evaluation

Figure 1: Visual representation of the Dick and Carey model used for most of this ID Document

Note: The prototype lacks the data required to analyse real instruction or entry behavior; when used, assumptions are made based on provided documentation and our own COS214 experiences. All revision stages and actual evaluation will also be skipped but will have hooks in place for easy implementation.

1 Stage One: Instructional Goals

By the end of this unit the instruction will have :
(All coding done in C++; all diagrams are UML Class Diagrams)

1. Introduced the problem of creation variability in OO code and why a creational pattern is needed.
2. Presented the Factory Method pattern in its canonical form (intent, forces, typical consequences).
3. Explained the canonical structure & terminology (Creator/ConcreteCreator, Product/ConcreteProduct) through UML class diagrams.
4. Explained the canonical structure & terminology (Creator/ConcreteCreator, Product/ConcreteProduct) through code examples.
5. Shown common pitfalls & misconceptions in code (e.g., ”wrapper around new”, client constructing concrete types, misuse of switch).
6. Demonstrated correct C++ realisation of FM in small examples (including lifecycle concerns that matter in FM implementations).
7. Situated FM among related patterns at recognition level (esp. Abstract Factory and Strategy) so learners know when FM is appropriate.

8. Provided guided C++ practice opportunities that culminate in a small refactor of naïve code to FM and a brief transfer d iscussion.

2 Stage Two: Instructional Analysis

Purpose. Identify the enabling knowledge, skills, behaviors, and conditions learners need in order to engage with this unit and complete the practice activities in the app.

2.1 What learners need to take part

A. Cognitive prerequisites from prior modules (assumed)

- Basic C++ syntax and control flow up to COS110 level.
- Classes, objects, headers vs implementation files, simple compilation.
- Pointers and references at an introductory level.
- OO coding and Inheritance at a basic level.
- Read simple UML class diagrams at a basic level (class boxes, generalisation, dependency arrows).

B. Digital and tool skills

- Use a modern web browser with JavaScript enabled.
- Type code into an in-browser editor, copy and paste, navigate tabs.
- Run unit tests in the app and read console style results.
- Upload or download small files if requested (not required for core flow).
- Basic keyboard navigation. Mouse or trackpad available.

C. Study and self-management skills

- Sustain attention for short micro-tasks of 5 to 8 minutes.
- Manage retries and review immediate feedback before moving on.
- Allocate at least one focused session of 45 to 90 minutes to complete the unit.
- Take brief notes from the cheat sheet or narration if helpful.

D. Motivation and attitudes

- Willingness to use an app for learning and assessment.
- Openness to immediate automated feedback and to trying again after a failed attempt.
- Curiosity about how and why the pattern is used.

E. Accessibility and learning preferences Ability to choose a preferred modality. The app provides:

- Visual: annotated UML, short animations where helpful.
- Auditory: narrated micro-explanations with full captions and transcripts.
- Read/Write: concise notes and checklists.
- Kinesthetic: hands-on code tasks with instant test feedback.
- Keyboard-only navigation supported. High-contrast theme available. Screen reader landmarks and ARIA live regions present on result panels.

F. Environment and equipment

- Laptop or desktop recommended. Minimum 1366×768 resolution.
- Stable internet connection suitable for loading short pages and test runs. The app is light on bandwidth.
- Headphones recommended for narrated clips in shared spaces.

2.2 What learners may not yet have and how we accommodate

- Confidence with UML reading may vary. The app provides a one-page UML cheat sheet and a labelled example for reference.
- C++ lifecycle details may be rusty for some students. The app includes short reminders where these affect the exercises.
- Attention span and time availability differ. Lessons are chunked into micro-tasks and can be completed in multiple short sessions.
- Bandwidth and noise constraints differ. Captions and transcripts are available, and all essential content has a text-first version.

2.3 Onboarding self-check (non-blocking)

At first launch, the app presents a short self-check to help learners prepare:

- Can you run a sample test and read the pass or fail result.
- Can you identify a base and a derived class in a tiny UML diagram.
- Do you prefer visual, audio, read/write, or code-first learning. The app labels screens accordingly.

Learners who answer ”no” are shown a brief tip or a link to the prescribed material from prior modules. Progress is not blocked.

2.4 Risks and mitigations

- Low confidence in UML
  - Mitigation: labelled examples, cheat sheet, and side-by-side code and diagram views.
- Limited time
  - Mitigation: micro-tasks that save progress automatically, resume where you left off.
- Limited bandwidth or noisy environment
  - Mitigation: captions, transcripts, and light pages with minimal media.

Note: This stage does not re-teach content. It specifies the enabling conditions and skills learners bring to the unit and how the app supports varied needs and preferences so that they can participate effectively.

3 Stage Three: Learner Characteristics

3.1 Prerequisite modules (in order)

- COS132 Imperative Programming (C++)
- WTW114 or equivalent mathematics
- COS110 Introduction to Program Design (C++)
- COS212 Data Structures and Algorithms (Java)

3.2 Expected entry knowledge and skills

- C++ foundations from the above modules.
- Understanding of classes, inheritance, and dynamic dispatch.
- Ability to read a UML class diagram.

We assume these have been achieved by all COS214 students. This unit does not re-teach them.

3.3 Variability and likely gaps

- Some learners may be utterly useless C++ coders.
- UML fluency varies. Some will need reminders on generalisation and simple dependencies.
- Confidence and preferred learning modes differ across the cohort.

3.4 Identification of gaps (onboarding self-check, non-blocking)

At first launch the app shows a short self-check:

- Run a sample test and read the pass or fail result.
- Identify base vs derived in a tiny UML diagram.
- Indicate preferred learning modes through VARK questionnaire.

Results are used to surface tips and references to previous modules required text. Progress is never blocked.

3.5 Support if something is missing

- Direct links to prescribed material from prior modules for refreshers.
- Inline micro-reminders where lifecycle rules affect an exercise.
- Hints keyed to common errors, available after one or two attempts.
- Resume anytime, with autosave.

3.6 VARK Analysis

Students will confidentially answer an up to date VARK survey to help them quantify how they best learn. This will in turn allow them to make use of the appropriately labeled in app features that align with their learning styles. As far as possible all lessons will have a full approach aligned with each preferred learning style.

4 Stage Four: Learning Outcomes (Performance Objectives)

Below are the four main learning outcomes and the sub-outcomes that define them.
Note: All sub outcomes are paired with the BLOOM level at which they will be summatively assessed and all code is in C++ and all diagrams are UML Class diagrams.

After this unit the student will be able to:

4.1 Know and Understand the aspects of the Factory Method Pattern

1. State the name of the Factory Method pattern. (Remember)
2. State the classification of Factory Method (creational). (Remember)
3. State the high-level strategy FM uses. (Remember)
4. State/Explain the intent of Factory Method in a given context. (Remember/Understand)
5. Describe the canonical UML structure of FM. (Understand)
6. Explain the programming problem FM solves in plain language. (Understand)
7. Identify and map participants (Creator, ConcreteCreator, Product, ConcreteProduct) in a short code/UML snippet. (Understand)
8. List related patterns (e.g., Abstract Factory, Strategy) and explain why they're related. (Remember/Understand)
9. Given a context, decide whether to apply FM and briefly justify. (Analyse; Evaluate)

4.2 Know how to analyse and apply the Factory Method UML class diagram from given code or a given context.

10. Produce a correct UML diagram from a given C++ FM code snippet, with roles, generalisation, dependencies and multiplicities. (Apply)
11. Design an appropriate custom FM UML for a given context (choose where the factory method lives, name participants, sketch associations). (Apply/Analyse)
12. Scan a large UML diagram and identify all FM participants and their roles in context, or conclude that FM is not present. (Analyse)
13. Within any UML that contains FM where classes have non-standard names, label the factory method operation and each role (Creator/ConcreteCreator/Product/ConcreteProduct). (Analyse)
14. Given an arbitrary code snippet, classify it as a FM role or non-role and justify (e.g., ”this class overrides `make()` → ConcreteCreator”). (Analyse)
15. Distinguish Factory Method vs Abstract Factory in UML/code with reasons. (Analyse)

4.3 Know how to understand, analyse and apply the Factory Method pattern in C++ coding contexts.

16. Implement FM in C++ for a given problem. (Apply)
17. Infer code implementation details from a given FM UML (which class instantiates, where creation occurs). (Understand/Analyse)
18. Trace which factory override executes and name the product for a given call path. (Understand/Analyse)
19. Review a code snippet and decide whether or not it could possibly be part of a FM. (Analyse)
20. Apply essential C++ lifecycle rules relevant to FM (member-initialiser lists, base-ctor calls, virtual destructor when deleting via base). (Apply)
21. Refactor a naïve client to FM so the client no longer constructs concretes. (Apply/Analyse)
22. Extend an existing FM with one new product and matching overriding Creator without changing the client. (Apply)
23. Translate both ways between code and UML for FM with roles correct. (Understand/Apply)

4.4 Know how the Factory Method pattern interacts with other design patterns.

24. Combine FM with another pattern such as Memento to solve a problem in a given context while motivating choices. (Evaluate)

Instructional Goal to Learning Outcome Mapping:

Table 1: Instructional Goal to Learning Outcome Mapping:

Instructional Goal	**Learning Outcomes covered**
G1 Creation variability & why a creational pattern is needed	LO4, LO6
G2 Canonical FM & "client must not construct concretes"	LO1, LO2, LO3, LO4, LO9
G3 Canonical UML structure & notation; produce/read/locate FM in diagrams	LO5, LO7, LO10, LO11, LO12, LO13, LO17, LO23
G4 Canonical code structure & role cues in code	LO7, LO14, LO19
G5 Pitfalls & misconceptions (wrapper around `new`, type switches, tight coupling)	LO14, LO19, LO21
G6 Correct C++ realisation & UML↔code conformance (virtual factory returning *Product*, overrides build concretes, virtual dtor, etc.)	LO10, LO14, LO16, LO17, LO18, LO20, LO21, LO22, LO23
G7 Related patterns at recognition level (Abstract Factory, Simple Factory) & when FM is appropriate	LO8, LO9, LO12, LO15, LO24*
G8 Guided hands-on practice culminating in refactor & brief transfer	LO9, LO11, LO16, LO21, LO22

5 Stage Five: Formative Assessment

Purpose. After every 3 micro-learning tasks, the app delivers a micro-quiz (MQ) targeting the LOs covered by those tasks. MQs are short, adaptive, and give immediate, constructive feedback tied to specific error classes. Learners may optionally over-practice; only the first graded attempt per MQ counts toward the 30-mark cap.

5.1 Global rules

- Time & marks: Total formative $\leq$ 60 minutes, $\leq$ 30 marks.
- Difficulty mix per MQ: ∼70% medium/hard, ∼30% quick recall.
- Attempts: Unlimited retries for learning; only first graded attempt counts. Variants randomised (names, products, paths).
- Feedback: Every miss returns (a) what failed, (b) why, (c) a 1-click remedial link or hint, (d) a follow-up practice item. Hard clears trigger an encouraging message.
- Accessibility: Keyboard-first, alt text on UML, captions/transcripts.
- Analytics logged: {user_or_anon_id, mq_id, item_id, lo_ids[], pass_fail, attempts, time_ms, error_class, remedial_clicked}.
- Error classes (examples): client-still-constructs, wrapper-around-new, type switch present, wrong factory return type, missing virtual dtor, mislabelled UML role, AF vs FM confusion, Simple-Factory confusion.

5.2 Micro-quiz blueprint ($\leq$ 6 MQs)

Scoring and timing: All micro-questions have a maximum of obtainable 5 marks. Each MQ is designed to take roughly 8 to 10 minutes to complete.

Table 2: Micro-quiz blueprint (A). MQ = Micro-Quiz.

MQ	Description	Trigger (after these micro-tasks / practices)	LOs targeted
MQ1	Intent and recognition	"Why patterns?", FM intent, "client must not construct" [T1]	1 - 4, 6, 9
MQ2	Canonical UML roles	UML roles and notation; role labels [T2]	5, 7, 13
MQ3	Code ⇔ UML	Code→UML translation; find FM in larger diagram [T4, T6]	10, 12, 23
MQ4	Code cues and conformance	Role cues in code; lifecycle reminders [T5]	14, 17 - 20
MQ5	Refactor	Move creation to factory [T5]	21
MQ6	Extend and differentiate	Add product/creator; FM vs AF/Simple [T3]	12, 15, 22

Table 3: Micro-quiz blueprint (B). MQ = Micro-Quiz.

MQ	Item types (examples)	# Items
MQ1	MCQ (scenario intent), Short-Just (2 sentences), quick FITB intent phrase	5
MQ2	UML-Label (roles, abstract/return), drag markers, one tiny MCQ on notation	4
MQ3	UML-Build from 40-50 lines; UML-Scan "is FM present?"	3
MQ4	Code-Read classify (role/non-role + reason), trace override→product, Code-Fix lifecycle	4
MQ5	Code-Refactor (no client→Concrete; tests) + one "find the seam" MCQ	2
MQ6	Code-Extend (client unchanged), MCQ triage FM vs AF vs Simple	3

If a learner takes a different path, the app selects the next MQ matching the last three micro-tasks' LOs. Extra practice beyond ∼60 minutes updates mastery heatmaps only.

5.3 Item design snippets (representative examples)

- **MQ2 (LO5/7/13) – UML-Label (2 pts).** Drag Creator/ConcreteCreator/Product/-ConcreteProduct onto nodes; tick abstract on Creator; set factory op return Product.

Feedback (miss): ”Factory returns a concrete type. In FM, the base factory returns Product so clients depend on abstractions. See: ‘FM notation cheat-sheet’.”

- **MQ3 (LO10) – UML-Build (3 pts).** From given C++ (Creator with `virtual std::unique_ptr<Product> make() const = 0;`, two overrides), assemble classes, inheritance, and the factory op signature.
  Feedback (miss): ”Base factory not marked virtual/abstract in UML. Add {abstract} on Creator and an operation `make(): Product`.”
- **MQ4 (LO14/20) – Code-Fix (2 pts).** Add missing `virtual ~Product()` and convert member-initialisers.
  Feedback (hit, hard): ”Great! Catching lifecycle issues prevents UB. You cleared a hard item.”
- **MQ5 (LO21) – Refactor (5 pts).** Replace client `new` + `switch` with calls to `Creator::make()`. Auto-checks: no `#include "Concrete*.h"` in client; no `new Concrete*`; tests green.
  Feedback (miss): ”Client still includes Concrete headers. Move those into ConcreteCreators; keep client dependent on Creator/Product only. See ‘Refactor guide → Step 3’.”
- **MQ6 (LO22/15) – Extend (3 pts) + Triaging (2 pts).** Add `ConcreteProductB` + `ConcreteCreatorB` without touching client; then choose FM vs AF vs Simple for a short scenario and cite one cue.
  Feedback (triage miss): ”Scenario mentions families of related products → Abstract Factory, not FM. Quick recap: AF creates families, FM creates one product via override.”

5.4 Mastery & reporting

- Per-LO mastery bands: Green (mastered), Amber (partial), Red (needs work).
- Heatmap per LO and per strand (UML, code, recognition).
- Next-step suggestions: When Red/Amber, app surfaces 1–2 targeted micro-tasks and a small practice set (ungraded) for that LO.

5.5 Encouraging & constructive feedback templates

Constructive (miss): ”You selected a factory that returns a concrete. In FM the base factory returns Product so clients depend on abstractions. Revisit ‘Factory return types’ (2 min), then try a quick practice.”

Encouraging (hard hit): ”Nice! Your refactor kept the client free of Concrete includes and all tests passed. That’s the core FM invariant. Want to try a tougher variant?”

5.6 Additional Practical Assignment Used Formatively

Scenario: You are building a tiny reporting utility that formats a vector of integers into different textual formats. The client code must not depend on any concrete writer. Learners implement Factory Method in C++ so the client asks a Creator for a Product writer and calls `render`.

Learning Outcomes hit
LO10, LO11, LO12, LO13, LO14, LO16, LO17, LO18, LO19, LO20, LO21, LO22, LO23
(Recognition LOs LO8, LO15 are reinforced via one triage item bundled with this practical.)

Time and marks

Suggested time: 25–30 minutes
Marks: 10

(Students are provided with all code that does not form part of the Factory Method)

Provided Code:

```
include/
  product.hpp          // interface Writer (to review)
  creator.hpp          // base Creator (pure virtual make)
  // students add their concrete headers in src/...
src/
  client.cpp           // uses Creator&; must NOT include any Concrete*.h
  main.cpp             // optional manual run
tests/
  test_render_csv.cpp
  test_render_md.cpp
  test_client_structure.cpp
  test_extend_tsv.cpp  // locked until "extend" step
```

product.hpp

```cpp
#pragma once
#include <string>
#include <vector>
struct Product {
  virtual ~Product() = default;                     // lifecycle correctness
  virtual std::string render(const std::vector<int>& data) const = 0;
};
```

creator.hpp

```cpp
#pragma once
#include <memory>
#include "product.hpp"
struct Creator {
  virtual ~Creator() = default;
  virtual std::unique_ptr<Product> make() const = 0;  // factory returns
     Product
};
```

client.cpp

```cpp
#include "creator.hpp"
// No includes of any Concrete*.h allowed
std::string export_report(const Creator& exporter, const std::vector<int>& v)
   {
  auto w = exporter.make();        // dynamic dispatch to ConcreteCreator
  return w->render(v);             // client knows only Product API
}
```

Tasks learners complete

- Implement products. `CSVWriter` and `MarkdownWriter` implement `Product::render`.
- CSV: `1,2,3\n`
- Markdown list: `- 1\n- 2\n- 3\n`

- Implement creators. `CSVExporter::make()` and `MarkdownExporter::make()` return `std::unique_pt` to the correct concrete.
- Keep the client abstract. Do not include any `Concrete*.h` in `client.cpp`. Do not instantiate concrete types in client. No switch on output type in client.
- Extension step. Add `TSVWriter` and `TSVExporter`. Constraint: do not change `client.cpp`.
- UML check. Produce the minimal UML (in-app drag build) that matches your code: roles, inheritance, factory op signature.

Autograder checks and marks (10)

Table 4: Autograder checks and marks (10)

Area	Check	Marks
Structure	*Creator::make()* is virtual/pure and returns *Product* in base	1
Structure	*Product* has `virtual ~Product()`	1
Client independence	No `#include "Concrete*.h"` or `new Concrete*` or type switch in client	2
Behaviour	CSV rendering exactly matches expected	2
Behaviour	Markdown rendering exactly matches expected	2
Extend	Add TSV writer/exporter; client unchanged; tests pass	2

Static structure checks can be done with simple text scans in the prototype harness: search `client.cpp` for include of Concrete, `new` with a known class prefix, or switch (; and confirm base signatures via a reflection header test compiling against students' headers.

Feedback map (error class → message + next step)

- **client-still-constructs**
  "Client references a concrete class. In FM the client depends on Creator/Product only. Move construction into `ConcreteExporter::make()`." → link "Refactor guide: moving creation."
- **type-switch-present**
  "Type switch found in client. Replace branch with `Creator::make()` and polymorphic dispatch." → link "Why switches are a smell for FM."
- **wrong-return-type**
  "Factory returns a concrete in the base. The base factory must return Product so clients stay abstract." → link "Factory return types."
- **missing-virtual-dtor**
  "Product lacks a virtual destructor. Deleting via base is undefined; mark `virtual ~Product()`." → link "C++ lifecycle in FM."

- **uml-roles-mislabelled**
  ”UML labels do not match code: ensure Creator declares an abstract `make(): Product` and ConcreteCreator overrides it to construct a ConcreteProduct.” → link ”FM UML cheat sheet.”

Hard clears trigger: ”Great! Client stayed independent of concretes and all tests passed. That is the core FM invariant.”

6 Stage Six: Instructional Strategy

Constraints and principles:
Time budget for instruction: Theory $\leq$ 120 min and Coding $\leq$ 120 min. Formative + diagnostic assessments (MQ1–MQ6 and the practical) are defined in Stage five and do not consume this budget but are accounted for such that the total time spent on the FM $\leq$ 5 hours student time.

Each micro-lesson blends multiple VARK modes where feasible. When a lesson offers two modality variants for the same LO, learners choose one (overlap does not add time). Lessons use worked examples, retrieval prompts, interleaving UML and code, and immediate ungraded feedback in-flow (graded feedback belongs to MQs).

For Theory Micro-lessons:

Theory subtotal: 90 min (leaves 30 min buffer for pacing, questions, or a second-modality variant where needed).

Optional VARK variants that do not add time if chosen instead of the default mode:

- T2-V (diagram-first) or T2-R (short text checklist).
- T4-V (live UML) or T4-K (drag-drop interactive on laptops in class).
- T6 group board walk or solo paper highlight set.

For Coding Micro-Lessons:

Coding subtotal: 95 min (leaves 25 min buffer for pacing or a short enrichment).

Optional enrichment (fit in buffer if desired, not required):
C7 (10–15 min): FM with one related pattern at recognition level (e.g., FM + Strategy). Read a tiny code sketch and identify the boundaries. Targets LO24 lightly without adding heavy creation.

6.1 Theory micro-lessons ($\leq$ 120 min total)

Table 5: Theory micro-lessons — ID, title (+Gagné), outcomes, modes, time, and activities.

ID	Title (+ Gagné focus)	Primary LOs	VARK	Time	Activities (what happens)
T1	Why patterns, why FM *(Gain attention; state objectives; recall)*	LO4, LO6, LO9	V,A,R	10 min	2-min problem vignette showing creation variability; quick poll on "what smells wrong"; 3-min mini-lecture on FM intent and the "client does not construct concretes" rule
T2	Canonical UML roles and notation *(Present; guidance)*	LO5, LO7, LO13	V,R	15 min	Instructor walk-through of Creator/Product roles, abstract markers, return types; labelled example; 2 quick retrieval prompts
T3	FM vs Simple Factory vs Abstract Factory (recognition) *(Present; elicit performance)*	LO8, LO12, LO15	V,A,R	12 min	Three tiny scenarios; learners pick FM/AF/SF and cite one decisive cue; instructor debrief
T4	Code $\rightarrow$ UML mapping (worked example) *(Present; model; elicit performance)*	LO10, LO23	V,R,K	18 min	Instructor converts ~50 lines of C++ FM into UML live; pair 2-min practice: students place roles on a half-built diagram
T5	Smell detection and seams *(Present; guidance)*	LO19, LO21	V,R	12 min	Identify client `new`, type switches, tight coupling; highlight the "creation seam" to move into the factory
T6	Scan a big UML for FM (or conclude absent) *(Elicit performance; feedback)*	LO12, LO13	V,K	15 min	Teams highlight FM participants in a larger diagram or justify "absent" with one cue; brief debrief
T7	Retrieval sprint and summary *(Enhance retention; closure)*	LO1–3, LO5, LO7	A,R	8 min	Six fast prompts (name, class, strategy, intent phrase, role labels, return type rule); recap key rules

6.2 Coding micro-lessons (≤ 120 min total)

ID	Title	Primary LOs	VARK	Gagné focus	Activities (what happens)	Time
C1	FM contracts in C++ (interfaces and lifecycle)	LO14, LO20	V,R	Present; guidance	Show minimal Product and Creator with virtual destructor and `make(): Product`; short pitfalls demo	10 min
C2	Implement minimal FM from scaffold	LO16	V,K	Model; guided practice	Fill in `CSVWriter` and `MarkdownWriter` plus exporters; run local tests; instructor models 1st then learners complete	20 min
C3	Trace dynamic dispatch	LO18	V,K	Elicit performance; feedback	Given a call path, predict which override runs and which product is created; verify by running tests	10 min
— MQ3 trigger — After C1–C3 (Stage five MQ4)						
C4	Refactor: move creation to factory	LO21, LO19	K	Guided practice	Start with a naïve client that constructs concretes; learners move creation into `Creator::make()`; pass structure checks	20 min
— MQ5 trigger — After C4 (Stage five MQ5)						
C5	Extend without touching client	LO22	K	Guided practice	Add a new ConcreteProduct and ConcreteCreator; prove client unchanged; tests pass	20 min
— MQ6 trigger — After C5 (+ recognition triage) (Stage five MQ6)						
C6	Round-trip UML ↔ code conformance	LO17, LO23	V,K	Elicit performance; feedback; transfer	From a UML diagram, write only base and override signatures; from code, mark UML return types and abstract markers	15 min

6.3 Orchestration timeline (what runs when)

Theory block (90 min instruction)
T1 → T2 → T3 → MQ1 → T4 → T5 → T6 → MQ2 → T7.

MQ1 and MQ2 are formative and counted in Stage five time, not here.

Coding block (95 min instruction)
C1 → C2 → C3 → MQ3 → C4 → MQ5 → C5 (+ recognition triage) → MQ6 → C6.

The practical defined in Stage five is separate and optional during the formative window.

6.4 How each micro-lesson advances the outcomes

- Theory T1–T3 establishes intent, classification, strategy, and pattern discrimination (LO1–4, LO6, LO8, LO9, LO12, LO15).
- Theory T4–T6 builds UML fluency and recognition in larger contexts (LO5, LO7, LO10, LO12, LO13, LO19, LO21, LO23).
- Coding C1–C6 realise FM correctly in C++, maintain invariants, refactor and extend safely, and check UML↔code conformance (LO14, LO16–LO23).
- Optional C7 lightly introduces LO24 without exceeding second-year constraints.

6.5 Accessibility and VARK within lessons

Every micro-lesson includes at least two of: annotated visuals, concise text notes, short narration with captions, and a hands-on element. Keyboard-only paths and high-contrast styles are supported. Where two modality variants are offered for the same LO (for example T2 and T4), learners choose the one that fits their preference; only one counts toward time.

7 Stage Seven: Instructional Materials

Table 6: Lesson assets (Stage Seven)

Category	Items
Core references	COS214 "Tackling Design Patterns" site (Factory Method chapter; UML recap); Gaddis *Starting Out with C++* (latest ed.); links to prescribed textbooks of prerequisite modules.
Visual (V)	Annotated UML (SVG/PNG); one short call-flow animation for `Creator::make()` dispatch.
Auditory (A)	60-90 s narrated clips per micro-lesson with captions and transcripts.
Read/Write (R)	One-page notes and cheat-sheets (intent/forces, roles, FM vs AF vs Simple, refactor checklist).
Kinesthetic (K)	In-browser code sandboxes with scaffolds (`product.hpp`, `creator.hpp`, `client.cpp`) and fast unit tests; drag-to-build UML canvas.
Accessibility baseline	Alt text and long-desc on diagrams; keyboard-only paths; visible focus; ARIA for result panels; high-contrast toggle; scalable fonts.
Assessment materials	MQ bank with LO+Bloom tags and feedback templates; formative practical (headers, read-only client, unit tests, autograder checks, remediation messages).
Teacher/admin pack	Mini slide deck; run-sheet; traceability CSV; onboarding self-check; analytics stub (LO heatmap, time-on-task, error counts).

8 Stage Eight: Diagnostic Assessment

During all formative and summative assessment student performance will be stored per question; learning outcome and BLOOM level. This data is to be used by the lecturing team to adjust the prototype application and lesson plan as needed for this and future cohorts.

A survey should also be sent to the students directly to be answered anonymously, this survey would allow the students to provide input on how effective they feel the prototype had been on teaching them the specified learning outcomes and whether they preferred it over classical teaching approaches.

9 Stage Nine: Summative Assessment & Evaluation

This stage is usually used to reflect back on the teaching from the point of view of the instructor, using the diagnostic assessment's data, here the aspects of the prototype that performed well would be kept but those that underperformed either in student engagement or learning outcome proficiency obtained.

Instead we describe a summative assessment plan for this chapter that would be included in the modules semester tests and exam.

Summatively assess all students confirming whether or not they were able meet the learning outcomes. (Would be assessed during a practical)

Summative Question-Type Library (aligned LOs & BLOOM)

#	Question type	Bloom	Hits these LOs	What the student does (practical)	Example stem (sketch)	How to score (auto/rubric)
1	Intent MCQ / FITB	Remember / Understand	1–4, 6	Choose/complete the canonical name, class (creational), intent phrase, or problem statement.	”FM is a ____ pattern used to ____.”	Auto: exact/acceptable answers list.
2	Scenario decision + 2-sentence rationale	Analyse (+ tiny Evaluate)	4, 6, 9, 15	Given a small scenario, choose FM/AF/Simple/Other and justify in $\leq$2 sentences.	”Given this plugin loader…choose the best pattern and justify one cue.”	Auto for choice; 0–2 rubric for rationale (cue cited, correct).
3	UML-Label (roles & markers)	Understand	5, 7, 13	Drag labels (Creator, ConcreteCreator, Product, ConcreteProduct), tick abstract, set factory op return type.	”Label the roles and mark the factory `make(): Product`.”	Auto: $\geq$90% labels/markers correct.
4	UML-Build from code	Apply	10, 23	From ∼50–60 lines of C++, place classes, generalisation, factory op signature/return, dependencies.	”Assemble the UML that matches this code.”	Auto: check role placements, inheritance, op signature.
5	UML-Scan (find/deny FM)	Analyse	12, 15	Click/select classes in a big diagram that form an FM; or answer ”not present” with a cue.	”Highlight the FM participants, or choose ‘absent’ and say why.”	Auto: set membership; rationale 0–1 if ”absent”.
6	Design a custom FM UML (from brief)	Apply / Analyse	11	Read a short context, sketch where the factory op lives, name roles, add associations.	”For the Exporter brief, place the factory and name participants.”	Rubric (3–4 pts): factory on Creator, returns Product, roles plausible.

#	Question type	Bloom	Hits these LOs	What the student does (practical)	Example stem (sketch)	How to score (auto/rubric)
7	Code-Read role classification	Analyse	7, 14	For 4–5 snippets, select {Creator, ConcreteCreator, Product, ConcreteProduct, Not FM} and give one-phrase reason.	”This class overrides `make()` → it is . . . because . . . ”	Auto on role; reason 0–1 keyword (override/abstract/return).
8	Conformance checks (True/False + fix)	Analyse	14, 17, 20, 23	Mark statements about contracts; optionally edit the wrong line (return type/virtual dtor).	”Base factory returns `ConcreteA*` (T/F). If false, correct the code.”	Auto: T/F + diff on small patch.
9	Code-Trace (dispatch → product)	Understand / Analyse	18	Predict which override runs and which concrete is built for a given call path.	”Given this call, the product type at runtime is ____.”	Auto: exact product name expected.
10	Code-Fix lifecycle	Apply	20	Add `virtual ~Product()`, correct member-initialiser order, base-ctor call.	”Make this compile & pass the lifecycle tests.”	Auto: tests/lint flags pass.
11	Refactor: move creation into factory	Apply / Analyse	21	Remove client `new`/`switch`; call `Creator::make()`; keep client abstract.	”Refactor so client has no concrete includes or `new`.”	Auto: structure grep + tests (no #include ”Concrete”, no `new Concrete`, tests green).
12	Extend: add product & creator	Apply	22	Add `ConcreteProductB` + `ConcreteCreatorB`; client unchanged.	”Add TSV writer and exporter without touching client.”	Auto: tests; file hash/AST confirms client untouched.
13	Code-Outline from UML	Apply	17, 23	Write only the base/override signatures from a UML diagram (no bodies).	”Write `Creator::make()` and the overrides’ signatures.”	Auto: signature match.
14	Translation duet (code ⇔ UML)	Understand / Apply	10, 17, 23	Part A: label UML; Part B: outline code signatures.	”Label roles; then write the corresponding method signatures.”	Auto: both parts $\geq$80%.

#	Question type	Bloom	Hits these LOs	What the student does (practical)	Example stem (sketch)	How to score (auto/rubric)
15	Smell detection + seam pointer	Analyse	19, 21	Mark the smell (client new, type switch, tight coupling) and say where creation should move.	”Select the smell and target seam.”	Auto: smell class + seam location match key.
16	Pattern discrimination (FM vs AF vs Simple)	Analyse	8, 12, 15	Given 3 tiny vignettes, choose pattern and cite one decisive cue.	”Family chosen together → ____ because ____.”	Auto choice; 0–1 cue rubric.
17	Tiny implement-from-scaffold	Apply	16	Fill in missing bits so all tests for a minimal FM pass (factory override + product method).	”Complete the override and product.”	Auto: unit tests pass.

References

University of Pretoria 2025 Yearbook

Dick and Carey Instructional Model - Educational Technology; available at: https://educationaltechnology.net/dick-and-carey-instructional-model/

COS214 Tackling Design Patterns - Linda Marshall and Vreda Pieterse; available at: https://www.cs.up.ac.za/cs/lmarshall/TDP/TDP.html

Appendix A: ID Questions' Answers

What is instructional design? A systematic way to plan learning: decide where we are going (learning outcomes), how we get there (methods, media, sequence), and how we know we arrived (criterion-referenced assessment and analytics). In this unit: teach Factory Method (FM) well and collect actionable data to improve lessons and all forms of assessment.

What does an instructional designer do?

- Analyses learners, context, constraints (people, tools, organisation).
- Writes outcomes aligned to Bloom (Remember/Understand/Apply/Analyse, tiny Evaluate, no Create).
- Chooses strategies/media that fit outcomes and time.
- Designs criterion-referenced items and feedback that produce evidence and next steps.
- Builds accessible materials with VARK options and WCAG-aware UI.
- Orchestrates short prototype → feedback → refine cycles (SAM-style cadence).

Why is instructional design important? It creates alignment (outcomes ↔ activities ↔ assessment) and evidence (analytics and error classes) so students learn efficiently and instructors can justify choices and improve iteratively.

Instructional design theories (toolbox we draw from)

- Behaviorism (clear criteria, immediate feedback).
- Cognitivism / Cognitive Load (chunking, worked examples, retrieval).
- Constructivism / Social learning (explanations, peer reasoning).
- Dual Coding (UML + text).
- ARCS motivation (attention, relevance, confidence, satisfaction).

How do Gagné's principles impact our design? We script lessons with Nine Events: Attention (smell vignette), Objectives (LOs shown), Recall (C++ lifecycle cues), Present (UML + minimal code), Guidance (cheat-sheets, hints), Practice (micro-tasks), Formative (micro-quizzes with rule-based + optional GenAI explanation), Summative (timed practical; no GenAI), and Transfer (where FM applies in real projects).

What are Merrill's principles? Problem-centred learning with Activation, Demonstration, Application, Integration. We anchor each micro-lesson to a small, authentic FM problem and cycle A→D→A→I.

How do Merrill's principles differ from Gagné's? Gagné provides the lesson sequence; Merrill keeps tasks problem-centred. We use Merrill to choose authentic tasks and Gagné to script each task's flow. They're complementary.

Models (frameworks/methods) for instructional design:

- **Dick & Carey (primary):** goals → analysis → objectives → criterion-referenced items → strategy → materials → formative → summative.
- **ADDIE:** we explicitly answer where/how/how we know in the Introduction.
- **SAM:** used as an iteration cadence for prototyping—not the primary model.

How do learning theories, learning styles and motivation link to ID?

- **Theories drive tactics:** worked examples, retrieval, immediate feedback, micro-quizzes.
- **We provide VARK choices:** Visual (annotated UML, simple animations), Auditory (60–90 s narrations with captions), Read/Write (concise notes, rule cards), Kinesthetic (in-browser coding with instant tests).
- **ARCS** is addressed via smell vignettes (Attention), clear course and career relevance (Relevance), graded practice with retries/mastery heatmaps (Confidence), and constructive + encouraging feedback (Satisfaction).

Are SE and ID so different? In short: No. One could view an ID document as the SE document counterpart in teaching.

They're distinct yet interlocking in this context. ID defines outcomes, lesson flow, assessment, and analytics; SE builds the product that delivers them. In this assignment they are separate documents, tied by a traceability chain (Instructional Goals ↔ Learning Outcomes ↔ Micro-lessons ↔ Micro-quizzes/practicals ↔ Analytics & error classes).