What makes the following Samman Learning Hour a good exemplar of a lesson plan? (Design for Approval Testing)

Design for Approval Testing

Extra challenge with dates, process ids, things in random order.

- 2 min connect: is testability an architectural concern?
- 10 min concrete: analyse the problem
- 10 min concrete: Demo pre-comparison processing
- 20 min concrete: fix the problem your own way
- 10 min concept: Pre-comparison processing
- 5 min conclusions: Strategies for testability

Connect - is testability an architectural concern?

Is it acceptable to re-design your production code to make it more testable? What kinds of things might you want to change in your architecture? Have you changed anything in your architecture purely for the sake of testing?

Hopefully they will mention controlling the current time.

Concrete - analyse why the tests are failing and how to fix them

Use the C#, java, C++ or python version of Supermarket Receipt starting on the 'with_date' branch. All the tests are currently failing.

- Why are the tests failing?
- Come up with two or three strategies for how to make them pass again.

Don't implement the strategy straight away. Discuss in a group the best solution.

Demo - pre-comparison processing

Demo a solution that fixes the problem by setting the date in receipt to be a fixed, known date.

Concrete - fix the problem in your own way

Look again at the strategies you came up with. Pick one and implement it. Alternatively implement the same one that was demoed.

Concept - pre-comparison processing

There are several points you can fix the output to make it suitable to use for approval.

- Just before comparison. Use a regex or similar to modify the string.
- In the printer. Have it configurable so it doesn't print every field
- Before you hand the complex object to the printer, remove or modify problematic fields.

For documentation for using Scrubbers:

- Java
- C++
- Python doesn't support it yet

Conclusions

Which strategy do you prefer? Is it acceptable to change the production code to improve testability?

Justifications

This Learning Hour is a **strong exemplar of a lesson plan** for several pedagogical and technical reasons. Let's break down what makes it effective, particularly for software teams working with **Test-Driven Development (TDD)** and **approval testing** in mind:

---

1. Strong Alignment with Learning Principles

Feature	Why It Works
Connect–Concrete–Concept–Conclusion (4C) structure	This classic learning arc engages learners by connecting to prior experience, grounding them in reality, abstracting new ideas, and then solidifying understanding.
Variety of learning modes	Combines discussion, demo, hands-on coding, and reflection—catering to different learning styles.
Active learning	Learners engage with broken code and must collaboratively reason and solve it. This is far more effective than passive absorption.

---

2. Relevance to Real-World Testability Challenges

- The lesson addresses **non-deterministic data** (e.g., current dates, process IDs, randomness), which are classic pain points in automated testing and often seen in legacy or unstructured systems.
- It reinforces a **key principle** of approval testing: *you often need to normalize or "scrub" data to make tests repeatable and useful.*

---

3. Drives Critical Thinking About Architecture

- Starts with a **provocative question**: *"Is testability an architectural concern?"*
- Encourages discussion about modifying code for testability—**challenging the learner's assumptions** and introducing ideas like **dependency injection**, **test doubles**, or **pre-comparison processing**.
- Helps developers understand that **maintainable tests are part of maintainable architecture**.

---

4. Practical, Tool-Specific Instruction with Language Support

- Offers **implementation across Java, C++, and Python**, making it adaptable to various team contexts.
- Introduces **Scrubbers**, a powerful feature in ApprovalTests, with documentation links—boosting self-sufficiency.
- The hands-on section lets devs **experiment with different strategies**, reinforcing understanding through application.

---

5. Clear Learning Objectives & Measurable Outcomes

Goal	**Evidence of Success**
Understand what causes approval tests to fail with non-deterministic output	Learners can articulate the root causes (e.g., time, randomness)
Learn strategies to "scrub" or normalize test outputs	Learners implement and compare at least one strategy
Explore whether and how to modify production code to support testing	Learners engage in discussion and make design decisions in code
Evaluate the trade-offs of different testing strategies	Final reflection prompts teams to express preferences and rationale

---

6. Promotes Team Dynamics & Ensemble Learning

- Learners must collaborate to:
  - Diagnose the problem
  - Debate solutions
  - Implement one approach
- This ensemble model mirrors **real team workflows** and improves communication and shared reasoning around code.

---

7. Flexible, but Focused

- Though structured, the plan allows teams to pick their own strategy to fix the problem—**enabling autonomy and creativity**.

- Optional languages and tools allow teams to stick to their tech stack—**increasing accessibility**.

Summary: Why It's Exemplary

Criteria	**Rating**
Pedagogical soundness	High
Technical relevance	High
Hands-on engagement	High
Clear goals and feedback loop	High
Scalable and adaptable	High