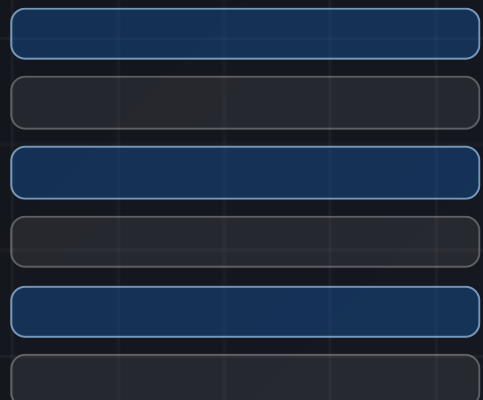


## GUIDE

# Thinking in Intended Design

## describing applications in the AI era

How to describe an application as business objects, rules, screens and the links between them, and how to work with AI on it.



CONTENTS

# What is inside

|           |                                                    |            |
|-----------|----------------------------------------------------|------------|
| <b>01</b> | <b>Why Intended Design is the skill now</b>        | Chapter 1  |
| <b>02</b> | <b>What to describe: six questions</b>             | Chapter 2  |
| <b>03</b> | <b>Concepts and relationships</b>                  | Chapter 3  |
| <b>04</b> | <b>Rules and behaviour</b>                         | Chapter 4  |
| <b>05</b> | <b>Screens and flows</b>                           | Chapter 5  |
| <b>06</b> | <b>Interdependencies: what depends on what</b>     | Chapter 6  |
| <b>07</b> | <b>Working with AI</b>                             | Chapter 7  |
| <b>08</b> | <b>Practice</b>                                    | Chapter 8  |
| <b>09</b> | <b>If you know UML or model-driven engineering</b> | Appendix A |
| <b>10</b> | <b>Glossary</b>                                    | Appendix B |

# Why Intended Design is the skill now

## What is in your head

Describe an order system out loud. You will say something like this: *customers place orders; an order has lines; each line is for a product; an order is open, confirmed or shipped; you cannot confirm an order if the stock is too low; staff see a list of open orders and open one to edit it.*

You did not mention a file, a class library or a line of code. You named **purposes** (take orders), **business concepts** (customer, order, line, product, status), **rules** (no confirmation without stock), **screens** (the list, the order) and **the links between them** (a line belongs to an order, an order belongs to a customer).

That is how people hold an application. It is also the thing that matters when something has to change: not where the code is, but what depends on what.

## What has changed

AI coding tools can now produce a lot of code quickly. Two things have not got easier:

- **Saying precisely what you want.** A tool can only build what has been described, and a loose description gives a loose result.
- **Checking what comes back.** More code means more to review, and a plausible change can break a rule that nobody told the tool about.

Both are skills of describing an application. To describe an application is to write down its concepts, rules, screens and relationships in a form that people can discuss and a tool can check.

## Why it helps AI, too

An AI that works from files has to infer your architecture from what it can read. An AI that works from an Intended Design is told what exists, how it is related and what is allowed. It can ask the Intended Design what depends on what, and its changes can be checked against the Intended Design before anything is built.

In zAppDev the engine produces the code from the Intended Design, and AI helps you build it. So the skill you practise is the same one the tool rewards: describing the application clearly.

## The Intended Design comes equipped

An Intended Design is not a blank page. The platform builds established patterns into it: ready-made components, all used in the same uniform way (bound to the view model, fed by a dataset, driven from the language), associations that state what happens on delete, permissions that protect actions, validation of the whole Intended Design, and an engine that applies an architecture consistently. You choose by name. Chapter 5 shows one example, measured.

## How this differs from memory, skills and specs

AI tools offer several ways to give an agent context: memory and instruction files, skills, written specifications, and search over the code. They help an agent find its way. They share one trait: they are text that someone wrote, and nothing checks the result against them. They can go stale, and an agent can follow them and still break a rule they never mentioned.

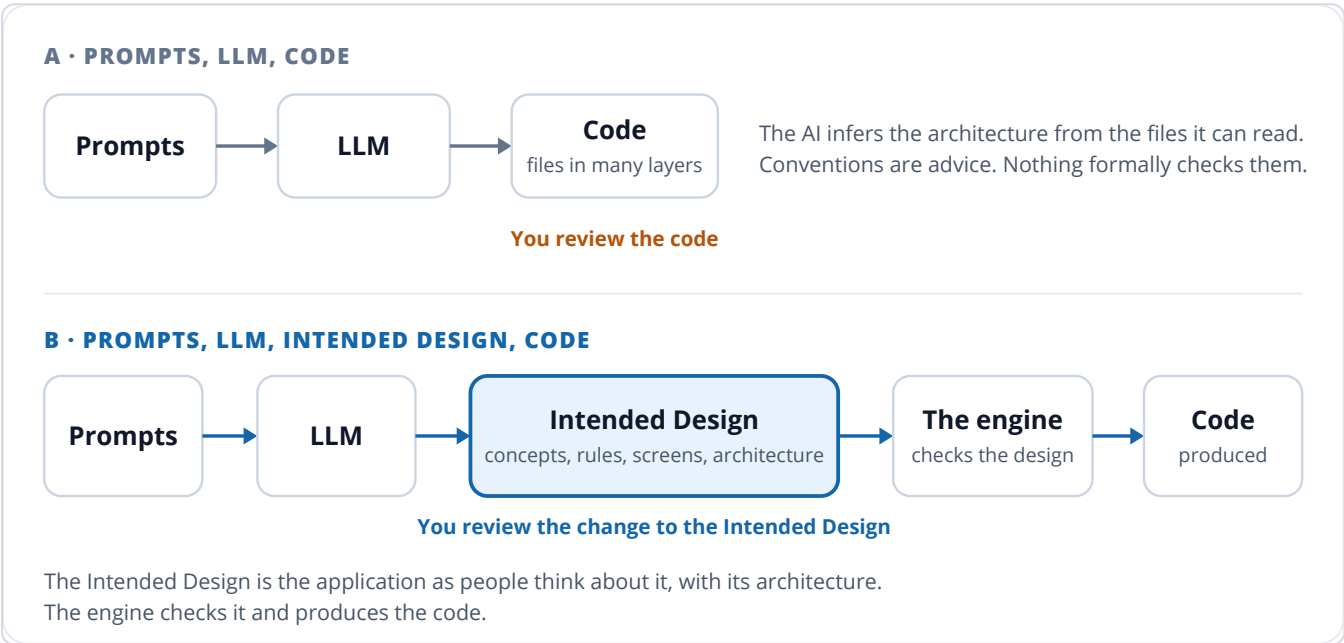
An Intended Design differs in kind. It is structured, so it holds elements and relationships and not paragraphs. It is connected, so you can ask what depends on what. It is checked: the engine validates the whole Intended Design before any code is produced, and the code comes from the Intended Design, so there is one source to keep in line.

These approaches work together. A skill can teach an agent how to describe an application well, and a specification can say what to build. The Intended Design is where the answer is kept and checked.

|                    | MEMORY AND INSTRUCTIONS | SKILLS                | SPECIFICATIONS                 | AN INTENDED DESIGN                            |
|--------------------|-------------------------|-----------------------|--------------------------------|-----------------------------------------------|
| What it is         | Text the agent reads    | Packaged know-how     | A description of what to build | Connected elements and relationships          |
| Checked by         | Nothing                 | Nothing               | Usually reviewers              | The engine                                    |
| When it goes stale | The agent does not know | Same                  | The code drifts from it        | Code is produced from it, so it stays in line |
| Best at            | Habits and conventions  | Repeatable procedures | Agreeing what to build         | Holding and checking what exists              |

### Two ways to work with AI

There are two common paths from a prompt to a working application.



In the first, an LLM turns your prompts into code. It works from the files it can read, so it has to infer the architecture and which conventions apply, and you review code.

In the second, the LLM turns your prompts into changes to an **Intended Design**. The Intended Design is the application as people think about it, and it includes the architecture: the views, the relationships between them and the choices the target applies. The engine checks the Intended Design before any code is produced, and you review the change to the Intended Design, in business terms.

Both paths use an LLM, and the code is real in both. They differ in what the AI works from, what is checked, and what you review. Chapter 7 follows the same request down both paths.

### What Intended Design is not

- It is not drawing diagrams for a document that drifts away from the software. An Intended Design is only useful if the software comes from it.

- It is not the right tool for everything. A novel algorithm or performance-critical code is easier to write as plain code. The Intended Design carries what is structured and repeated.
- It does not prove that your rules are right. A check can find a broken relationship or a wrong type. It cannot find a business rule that is simply wrong, so tests still matter.

## How to use this guide

Each chapter takes one part of the order example and shows how to think about it, what to write down, and the mistakes to avoid. The order example is a real, small application that we built in zAppDev for this guide: four classes, one enumeration, three associations, three rules, six tests and one screen. Boxes marked **Built in zAppDev** show what we actually made and what the tools reported. We validated, tested and built it. Chapter 7 puts it together with an AI agent. Chapter 8 has exercises. You can do them with pen and paper, or in Studio.

# What to describe: six questions

An application can be described in views. Each view answers one question, and keeping the questions apart keeps each one small.

| VIEW                 | THE QUESTION                     | WHAT YOU WRITE DOWN                   | IN THE ORDER EXAMPLE                                                        |
|----------------------|----------------------------------|---------------------------------------|-----------------------------------------------------------------------------|
| <b>1 Intent</b>      | Why does this application exist? | Goals, constraints, principles        | Take and ship orders reliably. Never ship what is not in stock.             |
| <b>2 Domain</b>      | What is the business made of?    | Business concepts and how they relate | Customer, Order, OrderLine, Product, OrderStatus                            |
| <b>3 Logic</b>       | What happens, and when?          | Rules and behaviour                   | Confirm an order only if stock covers every line                            |
| <b>4 UX/UI</b>       | What do people see and do?       | Screens, navigation, appearance       | An open-orders list and an order form                                       |
| <b>5 Integration</b> | How does it talk to the world?   | Services and external systems         | Orders offered as a service; a shipping provider called when an order ships |
| <b>6 Data</b>        | Where is it kept and moved?      | Storage, mappings, reports            | Orders stored in the database; a weekly sales report                        |

In zAppDev, the other views each have editors in Studio. Intent, the first view, is where you write down why the application exists. Keep it with the Intended Design, so that people and AI work against the same purpose.

## How to work through the views

1. **Start with the questions, not the screens.** The usual mistake is to begin with the screen that people see and let the concepts emerge by accident. Begin with purpose and concepts.
2. **Keep each view small.** If you cannot say what a view contains in a few lines, split the question.
3. **Name things the way the business does.** "Order" and "Customer", not "OrderRecord" and "CustomerTable". Names are the shared language between people and the AI.
4. **Write the links down.** An Intended Design is not a list of boxes. The value is in the relationships: a line belongs to an order, a rule uses an order, a screen shows an order.

## The links between views

The views are not separate documents. Elements link across them, and that is what makes questions answerable:

- the order form is bound to the **Order** concept;
- the confirm rule reads **OrderLine** and **Product**;
- the list screen runs the confirm rule from a button;
- the shipping call is made when an order moves to **Shipped**.

If **Order** changes, you can ask which screens, rules and services are affected. Chapter 6 shows how.

## Common mistakes

- Starting with screens and treating the data as an afterthought.
- Putting every rule in one place, so nobody can find what applies to an order.

- Using the names of tables and files instead of the names the business uses.
- Mixing questions: a rule that also describes a screen.

## Try it

Take an application you know and write one sentence for each of the six questions. Do not name any technology. If a view is empty, that is useful to know. If a view is overloaded, split it.

*Chapter 3 takes the Domain view and shows how to describe concepts and their relationships.*

# Concepts and relationships

The Domain view answers one question: **what is the business made of?** It is the part of the Intended Design that everything else leans on. Screens show it, rules use it, services offer it and the database stores it.

## Finding the concepts

Listen for the nouns the business uses, and ask of each one: *do we keep track of this?* In the order example: **Customer, Order, OrderLine, Product**. Each is a concept with its own identity: two orders are different even if their contents are the same.

A few cases need a decision:

- **A fixed list of choices** is an enumeration, not a concept. The order status (**Open, Confirmed, Shipped**) is one. Describe it as a list of allowed values, not as free text, so that a typo cannot create a fourth status.
- **A value that is calculated** is not typed in. The order total follows from its lines. Describe it as a value worked out from others, so it cannot disagree with them.
- **A detail that has no life of its own** is an attribute. The customer's e-mail address is a property of the customer, not a concept.

## Attributes

For each concept, write down what you know about it, and for each piece of information say its kind: text, number, date, yes or no, one of a list. Say which are required. Use the business's words: **Order date, Unit price**. Avoid storage names such as `ord_dt`.

## Relationships: say them in sentences

The value of an Intended Design is in the relationships, so write each one as a sentence first:

- *A customer places many orders. An order belongs to one customer.*
- *An order has many lines. A line belongs to one order.*
- *A line is for one product. A product appears on many lines.*

Then settle four things about each relationship. These are the decisions people most often leave unstated, and then an AI or a developer has to guess.

| DECISION                  | THE QUESTION                                   | CUSTOMER AND ORDER                                | ORDER AND ORDERLINE                                  | ORDERLINE AND PRODUCT                                    |
|---------------------------|------------------------------------------------|---------------------------------------------------|------------------------------------------------------|----------------------------------------------------------|
| <b>How many</b>           | One, zero or one, or many on each side?        | One customer, many orders                         | One order, many lines                                | One product, many lines                                  |
| <b>Who can reach whom</b> | From which side can you navigate to the other? | Both: a customer's orders, an order's customer    | Both                                                 | Line to product only. A product does not list its lines. |
| <b>On delete</b>          | What happens when one side is deleted?         | Protect: a customer with orders cannot be deleted | Delete together: deleting an order deletes its lines | Protect: a product on a line cannot be deleted           |
| <b>Stored or not</b>      | Is the link kept, or only held in memory?      | Stored                                            | Stored                                               | Stored                                                   |

Notice that "on delete" has real consequences. It is the difference between an Intended Design that protects your data and one that silently loses it.

### In zAppDev

The Business objects editor draws the classes, their attributes and the associations between them, and each association states exactly these settings: multiplicity on each side, navigability, what happens on delete, and whether it is persisted. Enumerations are defined next to the classes. A calculated value can be defined as a member that is worked out from other members.

**Built in zAppDev.** We described the order example in a business object called OrdersB0, with four classes (**Customer**, **Product**, **CustomerOrder**, **OrderLine**) and one enumeration (**OrderStatus**: Open, Confirmed, Shipped). We named the order class **CustomerOrder**, because *order* is also a keyword in database languages, and used **Order** as the role name on the line. The three associations, as the tool reports them:

| ASSOCIATION                 | HOW MANY                  | NAVIGABLE            | ON DELETE                                             |
|-----------------------------|---------------------------|----------------------|-------------------------------------------------------|
| Customer and CustomerOrder  | One customer, many orders | Both ways            | Restricted (a customer with orders cannot be deleted) |
| CustomerOrder and OrderLine | One order, many lines     | Both ways            | Cascade (deleting an order deletes its lines)         |
| Product and OrderLine       | One product, many lines   | Line to product only | Restricted                                            |

Validation of the Intended Design reported no errors. These settings are stored in the Intended Design. We did not test the delete behaviour at run time.

### Common mistakes

- **One big concept.** An "Order" that holds the customer's address, the product details and the shipping history. Split it by asking which things have their own identity.
- **Relationships left implicit.** Two concepts with a customer name copied into both. Link them, and keep the name in one place.
- **Status as free text.** Use an enumeration.
- **No delete rule.** If you have not decided what happens on delete, the tool will decide for you.
- **Storing what can be calculated.** A stored total will one day disagree with its lines.

### Working with an AI at this view

Describe the concepts and relationships in sentences, as above, and say the four decisions explicitly. An agent connected to the Intended Design can add classes, attributes and associations, and the Intended Design can be validated straight away. Review what it added against your sentences: the usual error is a missing delete rule or a relationship that cannot be navigated from the side you need.

### Try it

For an application you know, list its concepts. For each pair that is related, write the sentence and fill in the four decisions. Which delete rule surprised you?

# Rules and behaviour

The Logic view answers: **what happens, and when?** Rules are where applications go wrong in quiet ways, because a rule that is missing or misplaced looks fine until the wrong data gets through.

## Three kinds of rule

Ask which kind you are looking at, because each is written differently:

- **Must always be true.** *An order cannot be confirmed unless stock covers every line.* A condition that is checked.
- **Happens when.** *When an order is confirmed, reserve the stock and tell the customer.* A behaviour with a trigger.
- **Is calculated as.** *The order total is the sum of quantity times unit price.* A value derived from others.

## Put each rule next to the concept it is about

The rule about confirming an order belongs with **Order**, not in the screen that happens to have a Confirm button. Screens, services and automations then all ask the Order to confirm itself, and the rule exists once. When a rule lives in one screen, a second screen or a service that does the same job may not follow it.

## Keep each operation small, and name it as the business would

One operation, one rule, one verb: **Confirm order**, **Reserve stock**, **Calculate total**. State what it needs (an order that is open), what it does, and what it gives back or changes. If you cannot describe an operation in two or three sentences, it is doing too much.

Written as plain steps, the confirm rule might read:

**Confirm order.** Needs an order that is open. For each line, check that the product's stock covers the quantity. If any line fails, stop and say which line and why. Otherwise set the status to Confirmed and reserve the stock.

In zAppDev, operations like this are written in a typed language that is checked against the whole Intended Design. The same steps could be written in any language, and writing them as plain steps first is a good way to find out whether the rule is clear.

**Built in zAppDev.** The rule became an operation on **CustomerOrder** that returns a message, empty when it succeeds. The order total is a second operation:

```
function string Confirm() {
    if this.Status != Domain.OrderStatus.Open {
        return "Only an open order can be confirmed.";
    }
    foreach line in this.Lines {
        if line.Product.Stock < line.Quantity {
            return "Not enough stock for " + line.Product.Name + ".";
        }
    }
    foreach line in this.Lines {
        line.Product.Stock = line.Product.Stock - line.Quantity;
        line.Product.Save();
    }
    this.Status = Domain.OrderStatus.Confirmed;
    this.Save();
    return "";
}

function decimal Total() {
    decimal total = 0;
    foreach line in this.Lines {
        total = total + line.Quantity * line.Product.UnitPrice;
    }
    return total;
}
```

Validation reported no errors for either.

## Give every rule examples

A rule is only as clear as its examples. For each one, write a few cases as *given, when, then*:

| GIVEN                                         | WHEN    | THEN                                   |
|-----------------------------------------------|---------|----------------------------------------|
| <b>An open order, stock covers every line</b> | Confirm | Status is Confirmed, stock is reserved |
| <b>An open order, one line short of stock</b> | Confirm | Not confirmed, the short line is named |
| <b>A shipped order</b>                        | Confirm | Not allowed                            |
| <b>Two lines, 3 at 10 and 2 at 5</b>          | Total   | 40                                     |

These become tests. In zAppDev, unit tests are elements of the Intended Design with their own editor, and they run against the application. We wrote four, one for each row above, and all four passed: confirming reserves stock, a short line is rejected and named, a shipped order is rejected, and the total adds up. That matters because validation, as chapter 1 said, finds broken structure, not a wrong rule. Only a test can show that the rule does what the business means.

## Rules and screens

A screen should show a rule's result, not hold the rule. If the form needs to warn the user that stock is short, it calls the operation and shows what comes back. The same rule then applies when an order is confirmed through a service or by an automation.

For processes that run over time, such as *every night, release reservations on orders that were never confirmed*, the Intended Design has automation flows. They are the same idea: a named process, defined once and run when it is needed.

## Common mistakes

- **Rules in the screen.** Hard to find, and repeated.

- **Rules scattered.** The same condition checked in three places, slightly differently.
- **Vague rules.** "Handle stock properly" is not a rule. Write the condition and the outcome.
- **No examples and no tests.** The rule is then only as good as the memory of the person who wrote it.
- **Hidden order.** A rule that only works if another one ran first, and nobody wrote that down.

## Working with an AI at this view

Give the agent the rule as the plain steps and the examples, not as a wish. Ask it to place the operation with the concept it concerns, to write the tests from your examples, and then run them. Review the result by reading the examples against the steps. When a test fails, the failure points at either the rule or the example, and that is the useful part.

## Try it

Pick one rule from an application you know. Classify it (must always be true, happens when, or calculated as). Write it as plain steps and three given, when, then examples. Where would it live in the Intended Design?

*Chapter 5 turns to screens and flows: how forms are bound to the Intended Design.*

## Screens and flows

The UX/UI view answers: **what do people see and do?** It is the view people notice first, and the one that most often gets built first. Described well, it is the view that shows the rest of the Intended Design to people and changes the least about it.

### Start with people and tasks, not widgets

Before any screen, write down who uses the application and what each of them needs to get done.

| WHO         | WHAT THEY NEED TO DO                       | SCREENS AND FLOW                                    |
|-------------|--------------------------------------------|-----------------------------------------------------|
| Sales staff | Find an open order, correct it, confirm it | An open-orders list, an order form                  |
| Warehouse   | See confirmed orders and mark them shipped | A confirmed-orders list, a ship action on the order |
| Manager     | See sales over time                        | A sales report                                      |

Each row is a **flow**: a short sequence from what the person wants to what they get. *Open the list, choose an order, check its lines, confirm, return to the list.* A flow is easier to review than a screen, and it tells you which screens you need.

### What a screen holds

Think of every screen as answering three questions:

- **What does it show?** The data it needs, and where it comes from.
- **What can people do on it?** The actions it offers, such as save, confirm and delete.
- **Who may?** The roles and permissions that protect each action.

A screen is not a copy of a concept. The order form shows an order, but it also holds things that belong only to the screen: which tab is open, a filter on the list, a message to display. In zAppDev this is the screen's **view model**. Controls are bound to its members, and its actions call the operations you wrote in chapter 4.

### The common pattern: a list and a form

Most business screens are a pair. A **list** to find things, and a **form** to see and edit one. The list opens the form, the form saves and returns to the list. Because this pattern repeats for every concept, name it once and apply it the same way everywhere. Consistency is a feature: people learn the first screen and know the rest.

### Navigation and who sees what

A menu item is a link to a screen's action. If a person lacks the permission an action requires, the item should not appear for them. Describing permissions at the action and letting the menu follow them is safer than hiding things screen by screen.

### Keep appearance in the theme

Colours, fonts, spacing and component styles belong in a theme, shared by every screen. A screen that sets its own colours will drift from the others, and a rebrand then means editing every screen. In zAppDev the theme is its own element of the Intended Design.

### Design for the cases that are not the happy path

For every screen, ask what the person sees when:

- there is no data yet;
- a rule says no (the short line from chapter 4: show which line and why);
- they lack the permission;
- the data is large.

## In zAppDev

Forms are built in the Form editor from controls such as lists, tables, text boxes and buttons. A control is bound to a member of the form's view model. A button's *On Click* names an action, and a menu item opens a form's action. Actions can require permissions. The theme is edited separately. Validation reports bindings to members that do not exist.

**Built in zAppDev.** We built one screen for the order example, `OrdersList`, with the order's rules behind buttons. Our first version used a repeater and a small helper class to build each row by hand, because that is the pattern another screen in the same application used. We then rebuilt it with the platform's native **List** control, which is what a screen like this should use.

- **Data.** A dataset reads all the orders. There is no helper class and no code to build rows.
- **The list.** Columns for customer, date, status and discount. Sorting, paging, search and choosing columns come with the control.
- **Buttons.** Two list buttons act on the selected order: **Confirm**, which asks the order to confirm itself, and **Show total**, which shows the order total. The screen holds no rule of its own. Each button calls an action that calls the rule from chapter 4, and the answer goes into a note above the list.
- **Sample data.** Two customers, three products and three orders: one that can be confirmed, one with too little stock, and one already shipped, so every case of the rule can be tried.

Three things we learned:

- **The total is not a column.** A list shows members of a class, and the order total is an operation, not a member. We did not find a way to give an attribute a calculation through the tools we used, so the total is shown by a button and not in the list. Storing the total just to show it would contradict chapter 3.
- **A new form comes with an open door.** The new form had a default entry action that allows anonymous users. It showed nothing sensitive, but we removed it so that the only way in needs a signed-in user. A screen that anyone can open is a decision, not a default.
- **One harmless warning.** Validation reports that a standalone list sits with other controls, because we put a title and a note above it. The warning is expected for this layout.

Validation of the form reported no errors apart from that warning, the six tests passed, and the application built. We have not yet opened the rebuilt screen, so how it looks is untested.

## Repeater or List?

We built the same screen both ways in the same application, and measured both. The List is one example of many components that work this way. Every number is for this one example.

### THE SAME ORDERS SCREEN, TWO WAYS · IN THIS EXAMPLE

#### Lines of logic written



#### Lines generated



Generated lines per line of logic written: **about 40** for the custom build, **about 96** for the native List.

| IN THIS EXAMPLE                  | CUSTOM BUILD (REPEATER)                                                                                     | NATIVE LIST CONTROL                                                                          |
|----------------------------------|-------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------|
| <b>What you write</b>            | A helper class, a row-building operation, 46 lines of logic, 8 view-model entries and 8 hand-written styles | A dataset and a list with four columns, and 24 lines of logic                                |
| <b>What is generated for you</b> | The list as written: 1,863 lines                                                                            | The list, plus a server-side data source, saved views with sharing and download: 2,313 lines |
| <b>Changing a column</b>         | Four places                                                                                                 | One column                                                                                   |
| <b>The rules and tests</b>       | Six tests pass                                                                                              | The same six tests pass, unchanged                                                           |

In this example, 24 lines of logic produced 2,313 lines of generated code, about 96 for every line we wrote. The custom build produced about 40 for every line. Choosing the standard control meant half the logic, no helper class, and rules that did not have to change.

The extra generated code is the list's own behaviour: a server-side data source, saved views with sharing to users and roles, and file download. We checked that these methods are in the generated code. We did not test each one on the screen. More generated code is not better by itself. The point is that it is code you did not write or have to keep up to date.

A custom layout is still right when the standard control cannot express what you need, such as a card layout or an unusual row design. The custom build here had two things the list does not: a button on every row, and the total in each row. They cost a helper class and hand-written styles. The rule of thumb: use the standard control unless you can name what it cannot do.

### Common mistakes

- **A screen for every table.** Screens should follow tasks, not storage.
- **Rules in the screen.** Chapter 4 explained why.
- **Copy-pasted screens.** A change then has to be made many times.
- **Styling on each control.** Use the theme.
- **Forgetting roles.** A screen that anyone can open is a decision, not a default.
- **No empty and error states.** These are what people see first.

## Working with an AI at this view

Describe each screen on a small card: *purpose, who uses it, what it shows, what it can do, which rules it calls, which permission protects it*. An agent can create the form, add and bind the controls, and wire the actions. Then validate, and look at the result in the running application: a check can show that a binding is valid, only you can say whether the screen helps the person who uses it.

## Try it

Take one task from an application you know. Write it as a flow of four or five steps, then fill in a screen card for the main screen. Which permission protects it?

# Interdependencies: what depends on what

Chapters 3 to 5 described the parts. This chapter is about the links between them, which is where an Intended Design earns its keep. The question changes from *what is this?* to **what would a change touch?**

## Three questions

A connected Intended Design can answer three questions that a folder of files cannot answer reliably:

- **Where is this used?** Which screens show this attribute, and which operations write to it.
- **What would break?** If this changes, which forms, rules, services and reports are affected.
- **Is it still valid?** After the change, is the whole Intended Design consistent?

## Who points at whom

The links are specific and named. In the order example:

| FROM                       | TO                                | LINK             |
|----------------------------|-----------------------------------|------------------|
| <b>A form control</b>      | A member of the form's view model | Is bound to      |
| <b>A button</b>            | A form and an action              | Runs             |
| <b>An action</b>           | An operation on a concept         | Calls            |
| <b>An operation</b>        | Attributes of an order            | Reads and writes |
| <b>A service operation</b> | A concept                         | Uses             |
| <b>A permission</b>        | An action                         | Protects         |
| <b>A unit test</b>         | An operation                      | Checks           |

Follow the arrows and you can answer the first two questions for any element. Remove one and you can see the others that lose their target.

## An example: renaming a field

Rename **Price** to **Unit price** on a product. It sounds small. What depends on it?

- the product form, where a text box is bound to it;
- the order line, which copies it when a product is added;
- the total calculation, which multiplies it by the quantity;
- the product service, which returns it to other systems;
- the sales report, which sums it;
- a unit test that checks a line total.

Six places across four views, and some of them are in screens, not in code. A tool that searches only code will find the operations and miss the bindings.

*In one test of a real application, we renamed a class through an agent's tools. The rename did not update the places that referred to it, and validation then reported all eight operations that were broken. The Intended Design's search*

*tool covered code but not control bindings in that test, so validation and search were both needed. This was a single test on one application, not a measurement of how any tool performs in general.*

## A safe way to make any change

1. **Ask what uses it.** List the dependents before you change anything.
2. **Change it in one place, in the Intended Design.**
3. **Validate.** Broken links show up as errors with their location.
4. **Run the tests.** Validation shows broken structure. Tests show that the rules still do what the business means.
5. **Review the difference.** Read what changed, in terms of the Intended Design.
6. **Build and run.**

## Why views keep the blast radius small

If the interface only reaches the rules through named operations, changing a screen does not change a rule. If rules reach data only through the concepts, changing storage does not change a rule. Each separation is a boundary that a change does not cross. That is the practical reason for the six views: not neatness, but a smaller set of things to check each time.

## Working with an AI on a change

- **Ask for the list first.** "Before you change anything, list every element that uses Price." The answer is a checkable claim.
- **Require validation after every change.** And the tests.
- **Review the Intended Design difference, not just the application.** The difference says what the agent really did.
- **Do not trust a rename or a refactor you have not validated.** Tools can leave references behind.
- **Remember what a check cannot do.** It finds a broken link, not a wrong rule.

## Try it

Choose one attribute in an application you know. List everything that depends on it, in every view. Then check your list against the tool. What did you miss?

*Chapter 7 puts it together: working with an AI agent from brief to review.*

# Working with AI

Chapters 2 to 6 described what to describe. This chapter is about doing it with an AI agent: how to brief it, how to keep it honest, and how to review what it did. In zAppDev, AI helps you build the Intended Design and the engine produces the code from it. The skills below apply to any agent that works on an Intended Design, including zAppDev's own assistant and your own agent over MCP.

## The loop

Work in a short loop, and repeat it for each change:

- 1. **Read.** The agent reads the part of the Intended Design it will change.
- 2. **Change.** It makes one small change in the Intended Design.
- 3. **Check.** Validate the whole Intended Design, then run the tests.
- 4. **Build.** Build and look at the running application.
- 5. **Review.** Read what changed, in terms of the Intended Design.

Small steps matter more with AI than with people. An agent can make a dozen changes in the time it takes you to read one, and a long chain of unchecked changes is hard to untangle.

## Brief it in Intended Design terms

A good brief is a short description of the Intended Design, not a wish. For one change:

| PART OF THE BRIEF    | EXAMPLE                                                                       |
|----------------------|-------------------------------------------------------------------------------|
| Goal                 | Orders can have a discount, set by a manager                                  |
| Concepts             | Order gets a Discount percentage, between 0 and 100. Not required.            |
| Rules, with examples | Total is reduced by the discount. 10% on 200 gives 180. Over 100 is rejected. |
| Screens              | The order form shows Discount to managers only                                |
| Permissions          | Only the Manager role may change it                                           |
| Leave alone          | The shipping flow and the sales report                                        |

Notice that it uses the words of chapters 3 to 5, with examples for the rule and a statement of what must not change. A brief like this is easy for an agent to follow and easy for you to check against.

## Make it read first

Before it changes anything, ask the agent to say what it found: the concepts involved, the screens that show them, the rules that use them. If its summary is wrong, you have caught the problem before any change was made. Ask for the list of dependents too, as chapter 6 described.

## What to check, and what each check can tell you

| CHECK    | WHAT IT CAN TELL YOU                       | WHAT IT CANNOT          |
|----------|--------------------------------------------|-------------------------|
| Validate | Broken links, wrong types, missing targets | Whether a rule is right |

| CHECK                  | WHAT IT CAN TELL YOU                    | WHAT IT CANNOT                  |
|------------------------|-----------------------------------------|---------------------------------|
| Tests                  | That a rule does what your examples say | That your examples are complete |
| Build                  | That the whole application compiles     | That it does what people need   |
| Running it             | That the screen is usable               | That every case is covered      |
| Reading the difference | What the agent actually did             | Why it did it                   |

Use all of them. Each one catches what the others cannot.

### A real example

In one session we asked an agent to add a **Budget** to a project, using the tools of an application's MCP server. It read the project, added the attribute, and validation reported no errors. The agent then wrote a small rule that used the new attribute, and misspelt its name. The tool accepted the change when it was written. Validation of the whole Intended Design caught the misspelling and named the member that did not exist. The agent corrected it, the tests passed, and the application built. We then undid the change and built again.

Two points from it. First, the tool that wrote the change did not check it: the check came from validating the whole Intended Design. Second, removing an attribute through the tools left a reference to it in a screen's view model, so a cleanup had to be done by hand. *This was one session on one application. It shows what to look for, not how any agent performs in general.*

### The same request, two paths

Take one request: "Orders can have a discount percentage from 0 to 100. The total is reduced by it. Only open orders can change it." Here is what each path involves. The right-hand column is what we did, in the order example. The left-hand column is an illustration of how the same request goes when an agent works from code, not a measurement of any tool.

|                         | PATH A: PROMPTS, LLM, CODE                                                                   | PATH B: PROMPTS, LLM, INTENDED DESIGN, CODE                                                                                 |
|-------------------------|----------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------|
| What the AI starts from | The files it can read. It infers the architecture from them.                                 | The Intended Design: what exists, how it is related, which rules apply.                                                     |
| What it has to work out | Where orders live, how this code base does things, what else uses the total.                 | What depends on the total. The Intended Design answers: the screen shows the total through the rule, so it needs no change. |
| What it changes         | Typically several files across layers: the entity, data transfer objects, the screen, tests. | An attribute, a changed rule, a new rule and two tests.                                                                     |
| What you review         | A code diff across layers.                                                                   | The change to the Intended Design, in business terms.                                                                       |
| What is checked         | The compiler, and any tests that already exist.                                              | The whole Intended Design is validated, the tests run, then the engine produces the code.                                   |

**Built in zAppDev.** Path B, for real. The change was five elements of the Intended Design:

- a new attribute, **Discount**, a whole number on the order;
- the **Total** rule, changed to subtract the discount;
- a new rule, **SetDiscount**, which rejects a discount outside 0 to 100 and a change on an order that is not open;

- two new tests: ten percent off 200 gives 180, and 150 percent is rejected and leaves the total at 200.

```
function decimal Total() {
    decimal total = 0;
    foreach line in this.Lines {
        total = total + line.Quantity * line.Product.UnitPrice;
    }
    total = total - total * this.Discount / 100;
    return total;
}

function string SetDiscount(int percent) {
    if percent < 0 || percent > 100 {
        return "A discount must be between 0 and 100 percent.";
    }
    if this.Status != Domain.OrderStatus.Open {
        return "Only an open order can have its discount changed.";
    }
    this.Discount = percent;
    this.Save();
    return "";
}
```

Validation reported no errors. All six tests passed, the four from before and the two new ones, so the change did not break the rule from chapter 4. The application built. The orders screen needed no change, because it shows the total through the rule.

We left out the manager-only part of a real request. Restricting a rule to a role is a permission on the action, which we did not build for this example.

### What AI is good and less good at here

- **Good at:** the repetitive structure of an Intended Design (adding a concept and its screens in the usual pattern), keeping naming consistent, drafting rules and tests from your examples, and listing dependents.
- **Needs you for:** what the application is for, which concepts are missing, what a rule should be, what to call things, and whether a screen helps the person using it. An agent will write a plausible rule. Plausible is not the same as right.

### Using it together with memory, skills and specs

Chapter 1 compared the approaches. In practice you use them together, each for what it does well:

| USE                     | FOR                                                                        |
|-------------------------|----------------------------------------------------------------------------|
| Instructions and memory | Habits: naming, tone, what to ask before changing                          |
| Skills                  | Repeatable procedures, such as how to add a concept with its screens       |
| Specifications          | Agreeing what to build before you build it                                 |
| The Intended Design     | Holding and checking what exists, and the answer to "what depends on what" |

### Keep control

- **One token per agent**, limited to one application, with an expiry where you can, and revoked when no longer needed. Anyone with the token can change the Intended Design.
- **Review the Intended Design's changes** the way you would review a colleague's work.
- **Keep tests**, so a change that breaks a rule is found by a check and not by a user.
- **Know what the checks cover.** Say so in the team's definition of done.

## Common mistakes

- Handing over a whole feature in one prompt and reviewing only the result.
- Trusting a green validation as proof the rule is right.
- Not saying what to leave alone.
- Skipping the read-first step.
- Letting an agent clean up its own mistakes without looking at what it removed.

## Try it

Write the brief table for a change to an application you know. Ask an agent, or a colleague, to read first and summarise. How close was the summary to what you meant?

# Practice

Three exercises. Do them with pen and paper or in Studio. A free account is enough, and the Hello World tutorial in the documentation shows the editors. Answers and marking guidance will be in the teaching pack.

## Exercise 1: describe a library

A small library lends books to members. Members can borrow up to five books at a time. A book can have several copies. Staff record loans and returns.

1. List the concepts. Which are concepts and which are attributes? Which is an enumeration?
2. Write each relationship as a sentence, then fill in the four decisions: how many, who can reach whom, what happens on delete, stored or not.
3. What should happen when a member with a loan is deleted? Who decides?

## Exercise 2: a rule with examples

A late fee is charged when a book is returned after its due date: 20 cents per day, to a maximum equal to the price of the book.

1. Classify the rule: must always be true, happens when, or calculated as.
2. Write it as plain steps.
3. Write five given, when, then examples, including the maximum and a book returned on time.
4. Which concept does the operation belong to?

## Exercise 3: what depends on what

The library renames **Member** to **Borrower**.

1. List every element in every view that would be affected: forms, rules, services, reports, permissions and tests.
2. Write the six-step change from chapter 6.
3. Write the brief you would give an AI agent, including what it must leave alone.

## One-page checklist

**Six questions.** Why does it exist? What is it made of? What happens, and when? What do people see and do? How does it talk to the world? Where is it kept?

**For each concept.** Does it have its own identity? Are the attributes in the business's words? Is every relationship a sentence with the four decisions?

**For each rule.** Which kind? Plain steps? Three or more examples? Lives with its concept? A test?

**For each screen.** Which task and which person? What does it show and do? Which rule does it call? Which permission? What does it show when empty, or when a rule says no?

**For each change.** What uses it? Changed in one place? Validated? Tested? Difference reviewed? Built and run?

**For each AI brief.** Goal, concepts, rules with examples, screens, permissions, what to leave alone. Read first. Small steps. One token, scoped.

## If you know UML or model-driven engineering

If you have studied UML, model-driven engineering (MDE) or model-driven architecture (MDA), most of this guide will feel familiar, and some of it will be different. This appendix maps the ideas. It is a rough guide, and the correspondence is not exact.

| IN UML AND MDE                                          | IN THIS GUIDE AND IN ZAPPDEV                                                                                                                                    |
|---------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>A class diagram</b>                                  | The business objects: classes, attributes and associations, with multiplicity, navigability and delete behaviour stated                                         |
| <b>A metamodel</b>                                      | The shape of every Intended Design: the element types zAppDev knows and the named relationships allowed between them. An Intended Design is checked against it. |
| <b>A platform-independent model</b>                     | The Intended Design: what the application is, from the business side                                                                                            |
| <b>A model-to-code transformation</b>                   | The engine. It validates and optimises the Intended Design, and produces the code for a target stack.                                                           |
| <b>A platform-specific model</b>                        | Roughly, what a target decides: the architecture, libraries and patterns of the stack                                                                           |
| <b>Constraints and behaviour in a separate language</b> | Written inside the Intended Design, in the language, and checked against the whole Intended Design                                                              |
| <b>Round-trip engineering</b>                           | Not used. The generated code is overwritten on each build, and changes go into the Intended Design.                                                             |

### Where the approach differs

- **The Intended Design is the source.** In many MDE settings the model is a design that is kept alongside the code. Here the code is produced from the Intended Design, so there is one source to keep in line.
- **One connected Intended Design.** Data, screens, services, rules, themes and tests are parts of the same Intended Design, not separate diagrams.
- **The target is a program.** Choosing another target produces a different architecture from the same Intended Design.
- **AI works on the Intended Design.** The Intended Design is readable and changeable by an agent over MCP, and every change is checked before code is produced.

### What stays true

The long-standing lessons of MDE still apply: a model is only worth keeping if the software comes from it, models should be small enough to understand, and a model that tries to say everything says nothing. It is not the right tool for a novel algorithm or performance-critical code, which are easier to write as plain code.

# Glossary

- **Association.** A relationship between two classes, stating multiplicity, navigability, delete behaviour and whether it is stored.
- **Business object.** A class of the domain, with attributes, associations and operations.
- **Engine.** The program that validates the whole Intended Design and produces the code for the chosen target.
- **Enumeration.** A fixed list of allowed values, such as an order status.
- **Intended Design.** The application as people think about it: purposes, business concepts, rules and screens, and the links between them, written down so the engine can check it.
- **MCP (Model Context Protocol).** The protocol an agent uses to connect to a zAppDev application's server, to read, change, validate and build the Intended Design.
- **Operation.** A named piece of behaviour, such as Confirm order, written in the language.
- **Pattern.** A ready-made building block of the Intended Design, such as the List control or an association that states what happens on delete, which you choose by name.
- **Shape.** The element types and allowed relationships that every Intended Design follows.
- **Target.** One stack the engine produces for, through an implementation strategy: a program that encodes the architecture, patterns, libraries and security of the stack and produces code for it from the Intended Design.
- **Validation.** A check of the whole Intended Design for broken links and wrong types. It does not check that a rule is right.
- **View.** One of six questions about an application: intent, domain, logic, UX/UI, integration, data.
- **View model.** What a screen holds: the data it shows and its own state, to which controls are bound.