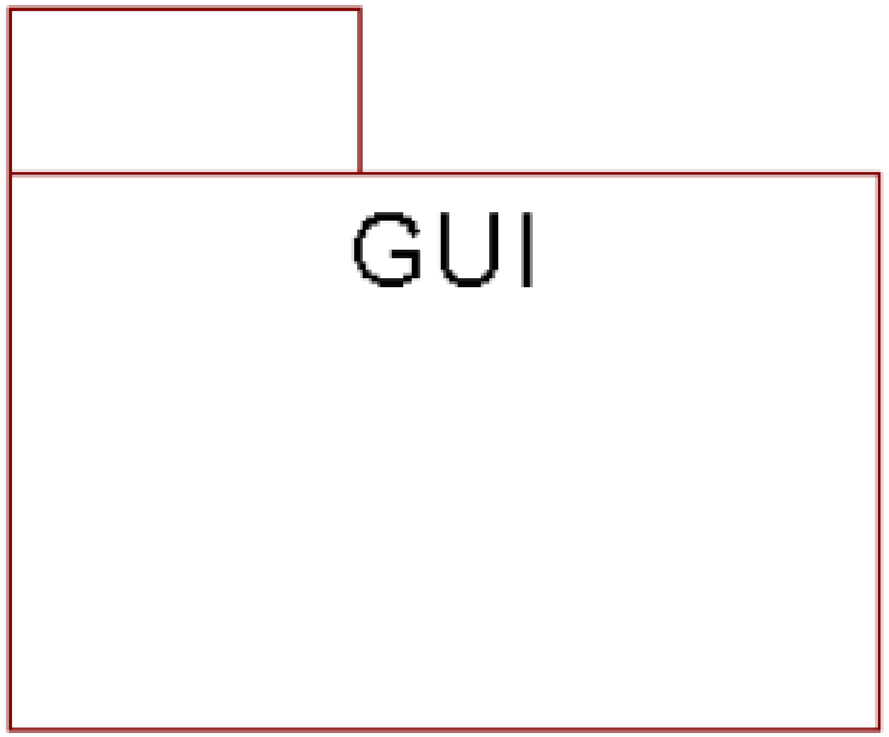
Lecture 10
System Architecture - Large and Complex Systems

we have considered relatively "small" systems. Generally, everything we have said so far would be easy to apply on a project with, say, 3 or 4 developers, with a handful of iterations lasting a couple of months each.

In this chapter, we'll have a look at some of the issues surrounding larger, more complex developments. Is the UML within an Iterative, Incremental Framework scaleable? And what else can the UML offer to help contain the complexity of such developments?

The UML Package Diagram

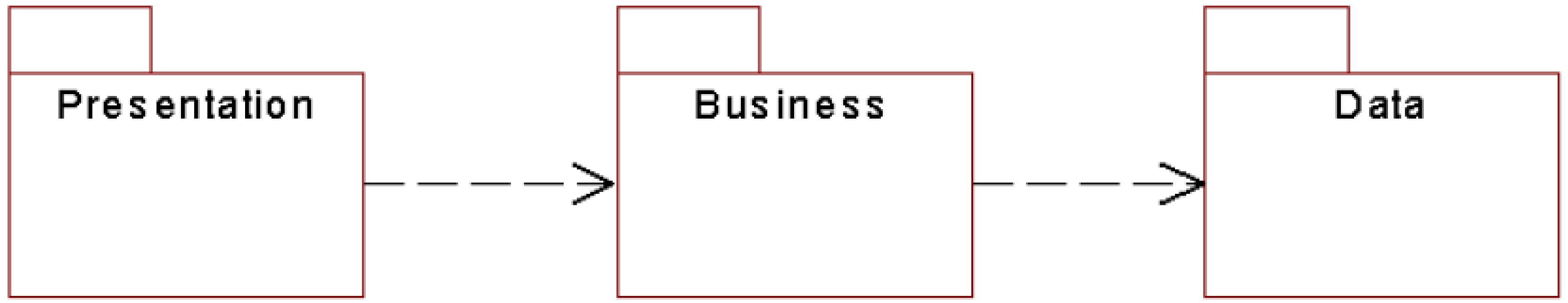
All UML Artefacts can be arranged into "UML Packages". A package is basically a logical container into which related elements can be placed - exactly like a folder or directory in an operating system.



Notation for a UML Package

In the above example, I have created a package called "GUI". I will probably be placing UML artefacts relating to Graphical User Interfaces inside the package.

We can display groups of packages, and the relationships between them, on the UML package diagram. The following is a simple example:



Three UML packages,

In the example above, I have notated the classic "three tier model" of software development. Items inside the "Presentation" package are dependent upon the items inside the "Business" package.

Note that the diagram does not show what is actually inside the package. Therefore, the Package Diagram provides a very "high level" view of the system. However, many case tools allow the user to double-click on the package icon to "open up" the package and explore the contents.

A package can contain other packages, and therefore packages can be arranged into hierarchies, again, exactly as with directory structures in operating systems.

Elements Inside a Package

Any UML artefact can be placed inside a package. However, the most common use of a package is to group related classes together. Sometimes, the model is used to group related Use Cases together.

Within a UML package, the names of the elements must be unique. So, for example, the name of every class within the package must be unique. However, one major benefit of packages is that it doesn't matter if there is a name class between two elements from different packages. This provides the immediate advantage that if we have two teams working in parallel, Team A does not need to worry about the contents of Team B's package (as far as naming goes). Nameclashes will not occur!

Why Packaging?

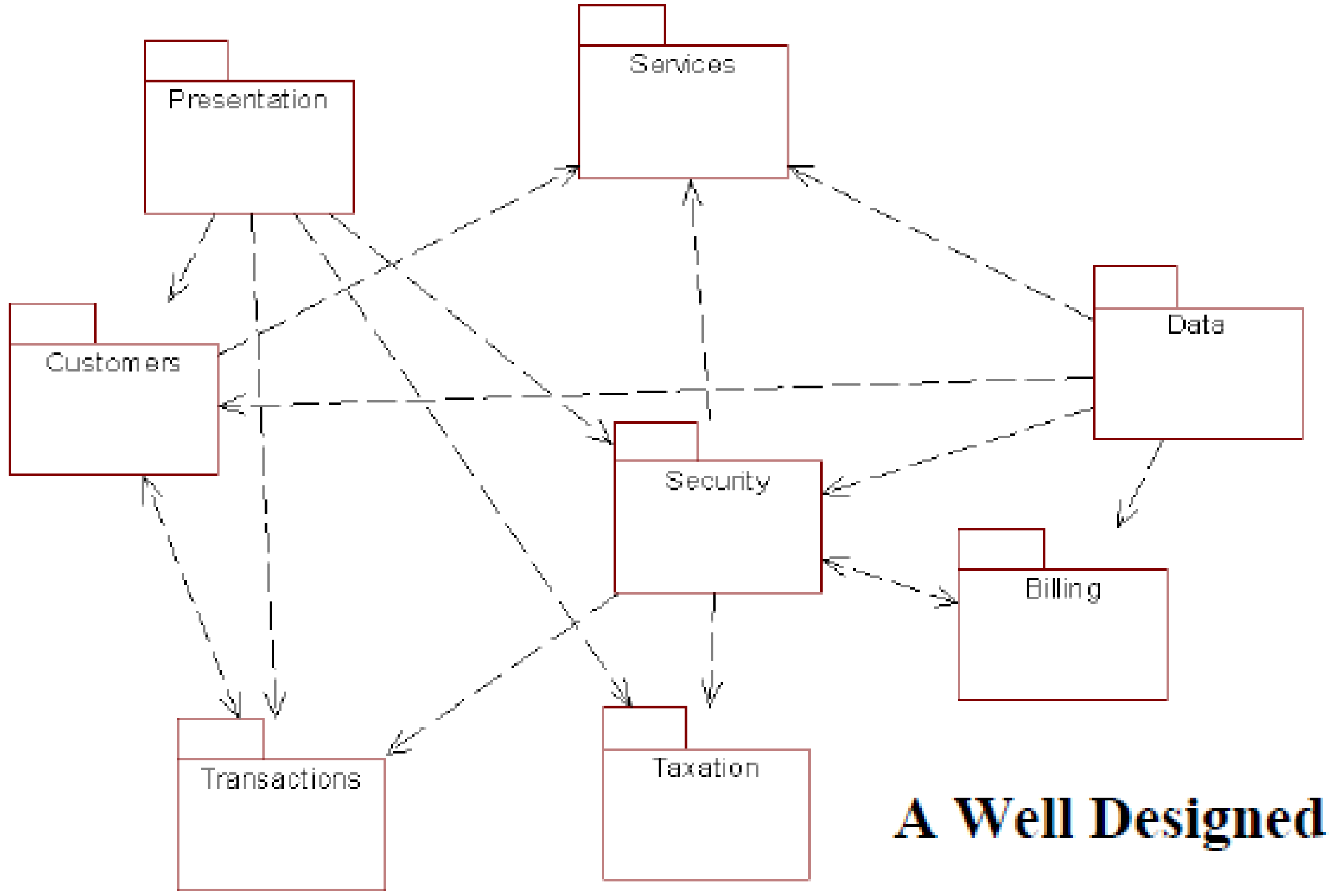
So why do we bother with packaging? Well, by careful use of packages, we can:

- Group large systems into easier to manage subsystems
- Allow parallel iterative development

Also, if we design each package well and provide clear interfaces between the packages (more on this shortly), we stand a chance of achieving code reuse. Reusing classes has turned out to be somewhat difficult (in some sense, a class is quite small and a bit fiddly to reuse), whereas a well designed package can be turned into a solid, reusable software component. For example, a graphics package could be used in many different projects.

Some Packaging Heuristics

Let us assume for this section that we are using the package diagram to partition classes into easy to understand and maintain packages.



A Well Designed Package Structure?

Several of the Heuristics from the GRASP chapter apply equally well to packaging. Three in particular stand out:

Expert

Which package should a class belong to? It should be obvious where each class belongs - if it isn't obvious then the package diagram is probably lacking.

High Cohesion

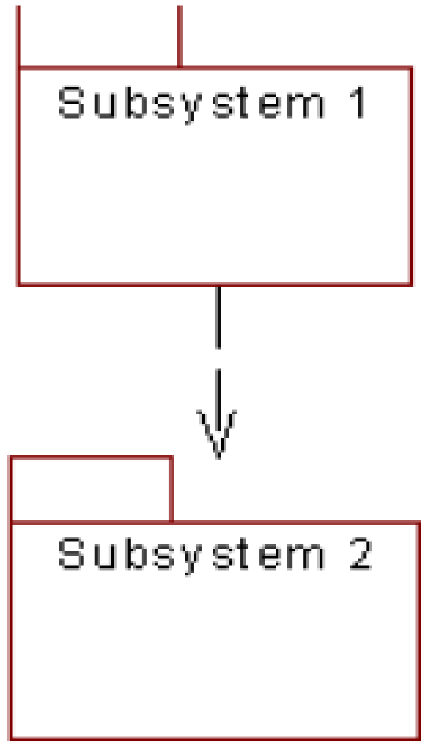
A package shouldn't do too much (or it will be difficult to understand, and certainly difficult to reuse).

Loose Coupling

Dependencies between packages should be kept to an absolute minimum. The diagram above is a made up example, but it looks fairly horrendous! Why is there so much cross package communication?

Handling Cross Package Communication

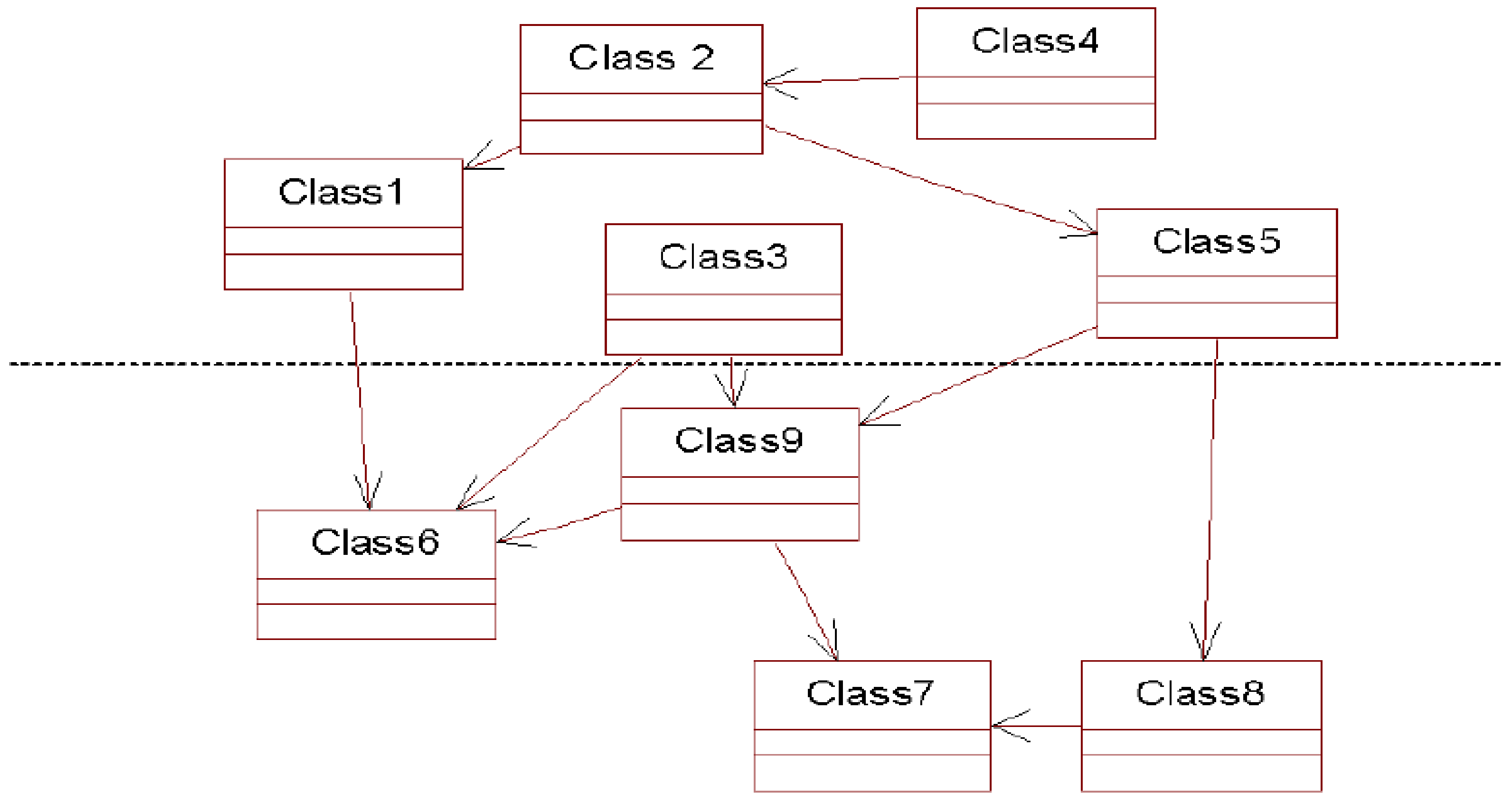
Assume we have two packages, each containing a number of classes.



Two Subsystems, modelled as UML Packages

From the dependency arrow, we can see that classes in the "Subsystem 1" package make calls to classes in the "Subsystem 2" package.

If we were to drill down and look inside the two packages, we might see something like the following (the attributes and operations have been removed for clarity):



Classes across two subsystems. The dashed line represents the subsystem boundary

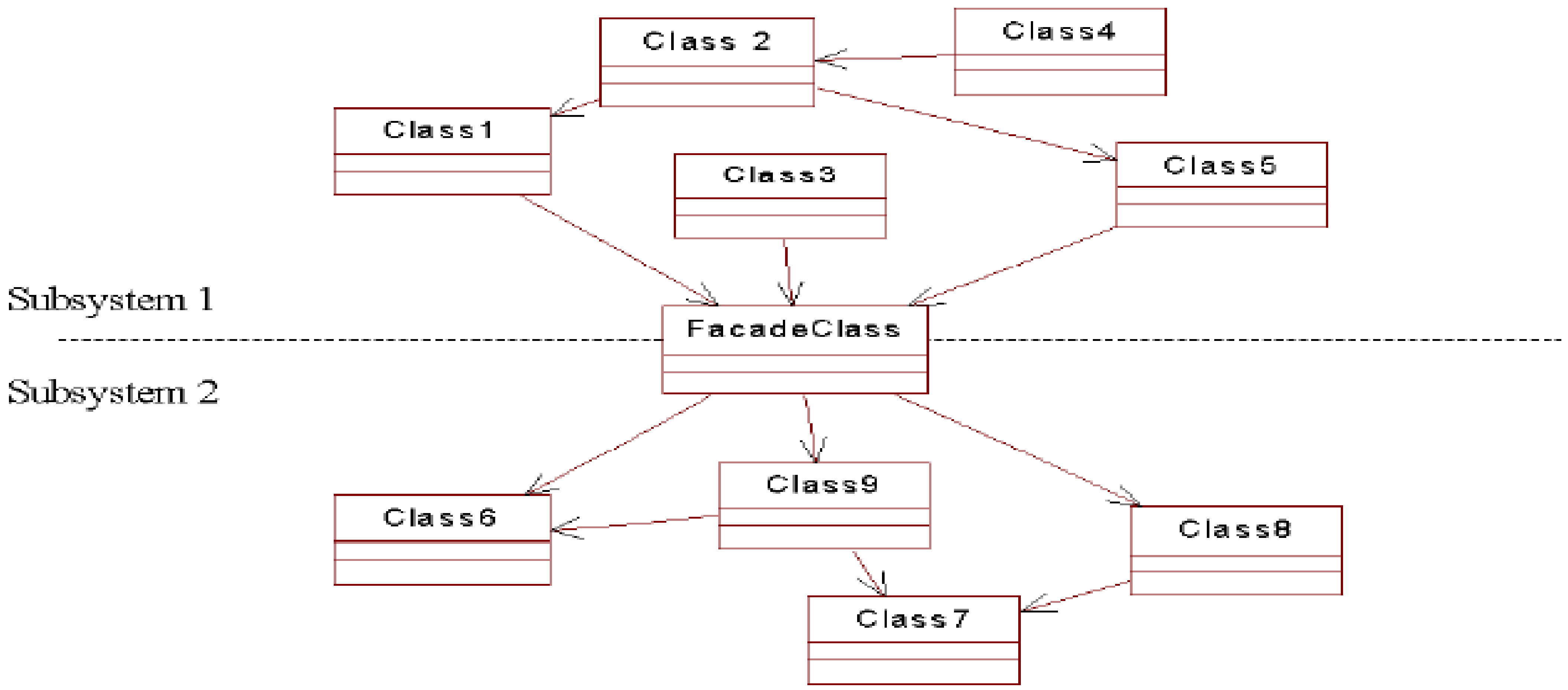
Basically, we have a situation where any class from the "Subsystem 1" package can call any class from the "Subsystem 2" package. Is this a good design?

Clearly, this idea leaves something to be desired. What if we needed to remove the Subsystem 1 package and replace it with a new subsystem (let's say that we are removing a terminal based user interface and replacing it with an all singing, all dancing graphical interface).

There would be a lot of work involved to understand the impact of the change. We would have to ensure that every class in the old subsystem is replaced with a corresponding class in the new subsystem. Very messy, and very inelegant. Luckily, there exists a design pattern called a Facade to help us with this problem.

The Facade Pattern

A better solution is to employ an extra class to act as a "go between" between the two subsystems. This type of class is called a Facade, and will provide, via its public interface, a collection of all of the public methods that the subsystem can support.



The Facade Solution

Now, calls are not made across the subsystem boundary, but all calls are directed through the Facade. If one subsystem were to be replaced, then the only change required would be to update the Facade.

The Java language has excellent support for this concept. As well as the usual Private, Public and Protected class visibilities, Java provides a fourth level of protection called **Package Protection**. If a class is designated as package rather than public, only classes from the same package may access it. This is a very strong level of encapsulation - by making all classes except the Facades **package**, each team building the subsystem really can work independently of each other.

Architecture-Centric Development

The Rational Unified Process strongly emphasises the concept of Architecture-centric development. Essentially, this means that the system is planned as a collection of Subsystems from a very early stage in the project development.

By creating a group of small, easy to manage subsystems, small development teams (maybe of just 3 or 4 people) can be allocated to each subsystem, and, as much as possible, can work in parallel, independent of each other.

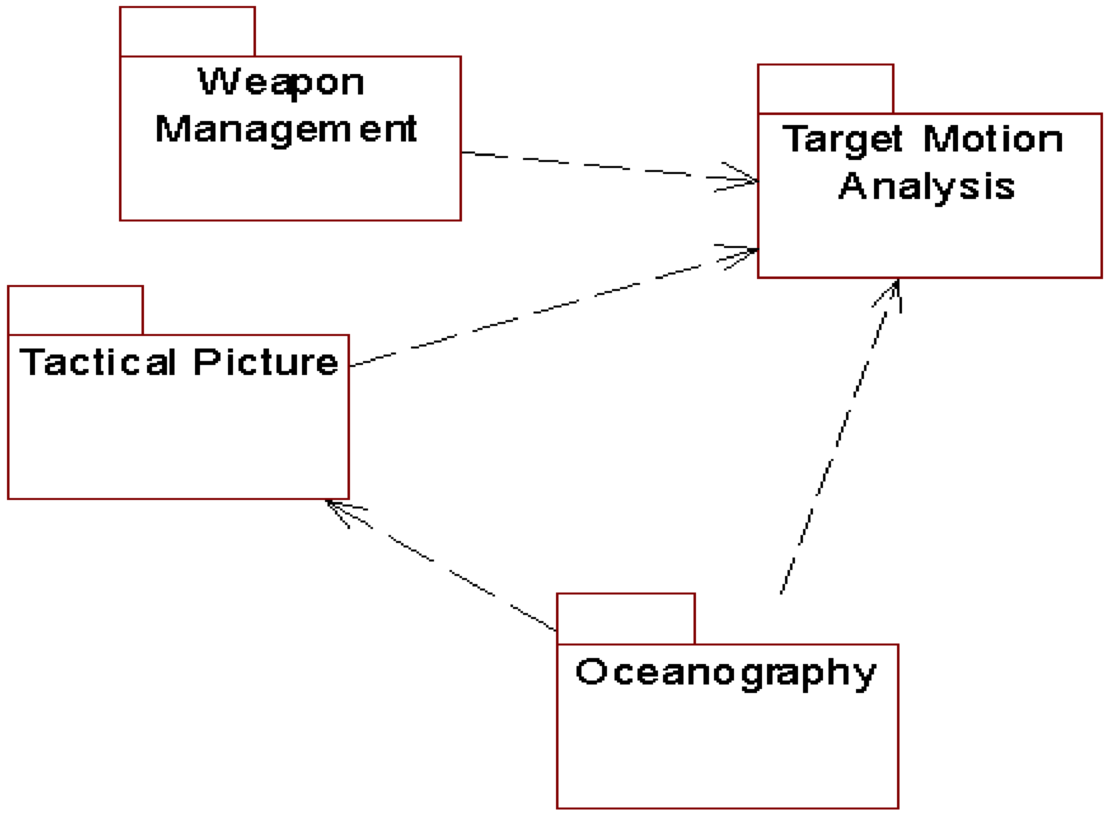
Clearly, this is far easier said than done. To underline the importance of this architecture activity, a **full time** architecture team is appointed (this could be a single person). This team is charged with the management of the architectural model - they would own and maintain everything related to the high level "big picture" of the system.

In other words, this team would own the package diagram. In addition, the architecture team would also own and control the interfaces (the Facades) between the subsystems. Clearly, as the project progress, changes will need to be made to the Facades, but those changes must be performed by the central architecture team, and not the developers working on individual subsystems.

As the architecture team maintain a constant "high level" view of the system, they are best placed to understand the impact changes to the interfaces between subsystems might have.

Example

For a major command and control system, the architecture team make a first cut of the system architecture by identifying the major areas of functionality to be offered by the system. They produce the following package diagram:

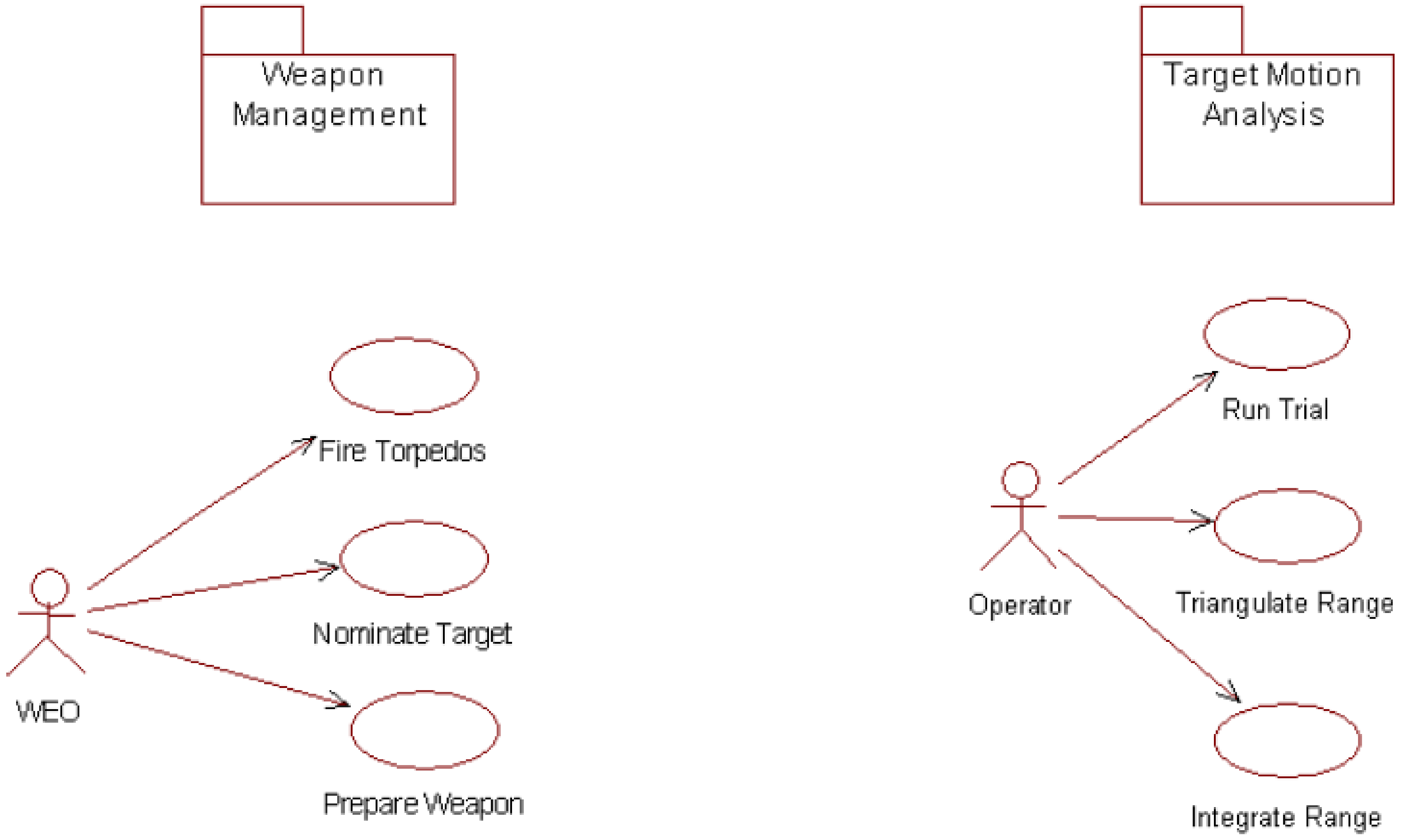


First Cut Subsystem Plan using a UML package diagram

Note that the architecture is not set in stone - the architecture team will evolve and expand the architecture as the project progresses, to contain the complexity of each subsystem.

The team would continue setting up subsystems until the size of each subsystem is not too complex, and is easy to manage.

Use Cases may well then be built for each subsystem. Each subsystem is treated as a system in its own right, exactly as we did in the early stages of the book:



Parallel Use Case Modelling

Handling Large Use Cases

Another problem with such large scale development is that these "first cut" use cases identified at the Elaboration phase may well be far too big to develop in a single iteration. The solution is **not** to make the iterations longer (this would cause complexity to rise again). Rather, the solution is break the Use Case into a series of easier to manage "versions".

For example, the "Fire Torpedoes" Use Case pictured above is identified, after the elaboration phase, to be a particularly large and difficult Use Case. The Use Case is therefore divided into separate versions, as follows:

- Version 1 - allows the opening of bow caps
- Version 2 - allows interlocks to be set
- Version 3 - allows the discharge of weapons

The aim is to ensure that each version is easy to understand, and achievable in a single iteration. So the Fire Torpedoes Use Case would take three iterations to complete.

The Construction Phase

The construction phase carries on as described in earlier chapters, but with each subsystem being developed, iteratively, by separate teams, working in parallel and as independently as possible.

At the end of each iteration, a phase of integration testing will take place, where the interfaces across subsystems are tested.

Summary

This chapter looked at some of the issues surrounding large scale system development. It is clear that although the UML is designed to be scaleable, transferring the Iterative Incremental Framework to large projects is far from a simple exercise.

The best approach at the moment seems to be the Architecture Centric approach proposed by Rational Corp:

- Define subsystems from an early stage
- Keep complexity as manageable as possible
- Iterate in parallel but don't hack interfaces
- Appoint a central architecture team

The package model provided by the UML provides a way of containing the large complexity, and this model should be owned by the architecture team.