Loosely-Coupled Sakai

Added by Ray Davis, last edited by Ray Davis on Jan 18, 2006

Loosely-Coupled Sakai

Ray Davis - University of California, Berkeley

(Based on a presentation at the Sakai Winter 2005 conference)

1. Introduction

In 2005, besides serving on the Sakai Core Architecture team, I participated in developing two new Sakai applications. Both were successful despite a number of frustrations. In this paper, I'll explain some of the frustrations and how we overcame them.

I've sometimes heard our approach described as "non-standard" or "extra work." Because of that, I'd like to begin with a project overview.

The **Sakai Gradebook** was one of the earliest adopters of the agreed-upon Sakai technology suite: Spring, Hibernate, and JSF. It was one of the first Sakai projects to include automatic unit test coverage and to be designed with the Sakai Style Guide in mind. It was the first Sakai application to support filtering by course sections and the first Sakai application to clean up stale data when a site is deleted. Not long before its initial release, a QA tester said she was worried about it because "there didn't seem to be enough bug reports."

The **Sakai Section Info** project built on the Gradebook's approach and achieved more:

- A relatively complex new JSF application for managing new Sakai framework capabilities, delivered *before* framework development was complete
- An LMS middleware package tailored to client needs, with full integration support for standalone testing
- Three complete implementations: standalone, Sakai 2.0 based, and Sakai 2.1 based

The projects were delivered on schedule with very small teams: two developers for the initial release of the Gradebook and only one full-time developer for Section Info.

It's true that both can be built as standalone applications. But clearly that didn't slow us down, make us inefficient, or keep us from fully integrating with Sakai. Instead, standalone build capabilities helped us meet our goals.

2. Empirical programming

Software engineers are hired to **deliver usable useful applications in a timely fashion.**

That statement sounds fairly innocuous. But take it seriously and a lot of specifics can be deduced.

To begin with, our development is pragmatic and *empirical* – that is, it's based on evidence. We cycle between two questions:

1. **Is it useful?** - Don't start work until you know what work needs to be done.
2. **Are you *sure* it's useful?** - Don't say you're done until you have evidence.

Despite its exorbitant expense, most software engineering ends in failure. To guard against false starts and wasted effort, we want to reduce the time between those two questions as much as possible.

Since the user determines what's useful, we need a cross-disciplinary team that includes one or more **user representatives**. That way, we can demonstrate early implementations, discuss problems when they arise, and negotiate solutions quickly.

We **code incrementally**, starting with basic useful functionality and adding to it without going backwards. That lets testing begin early, and, if the project is interrupted for some reason, we still have something to show for it.

Being mere mortals, we can't produce rapid results without making mistakes. That means we'll have to do things over. Rather than pretend that it's not going to happen, we count on a **cyclical** process. If we know something is going to be rethought, our goal is pure development speed to reduce the pain when we throw it away. JUnit test coverage makes such refactoring less nerve-racking, and the Spring framework makes JUnit testing much easier.

Lazy optimization is already a familiar idea to many programmers. Don't waste time trying to guess at how to make lower-level code more efficient. Allow flexibility for tuning in the high-level design and then gather evidence to discover where performance and scalability need work.

Lazy generalization doesn't get talked about as much. But just as good programmers have an urge to make code clean and efficient, they also have an urge to generalize and make code re-usable. Much of the time they do that prematurely, become attached to their beautiful premature generalization, and the project drags. Instead, we try to stick to specifics and wait for evidence that generalization is needed. Usually the first copy-and-paste is the signal.

Both of these are aspects of **opportunistic refactoring**. We piggyback design improvements on bug fixes and new feature development. The ideal (not often achieved!) is to check in less code when we deliver more functionality.

Finally, we try for **loose coupling** to external services.

3. "Loose coupling"

Loose coupling simply means that some standard software design principles used *within* a project also apply *between* projects. It's separation of concerns and centralization of concerns at a project level.

And the goals are the same: Focus on one doable task at a time. Improve maintainability by decreasing interdependencies, redundancies, and copy-and-paste logic. Improve testing. Preserve your schedule by avoiding interruptions.

As a buzz-phrase, loose coupling has been associated with several trendy technologies: web services, for example, and asynchronous document-centered design. In our case, we rely heavily on Spring's ability to inject interface implementations at runtime. Still, no matter how we picture loose coupling, we probably have a visceral idea of what "tight coupling" feels like.

But there's some ambiguity here. Sometimes "separation of concerns" becomes *replication* of concerns. Rather than taking advantage of standards, programmers have a tendency to think that their own code will outlast anyone else's. It's just human nature to overrate the centrality of what we're closest to. (And it's just marketing nature to try to lock customers into a proprietary approach.) Here are some real-life examples I've seen:

- A Java library which covered every single AWT class with a branded equivalent.
- A new Java interface for logging so that programmers wouldn't have to choose between the two existing standard Java logging approaches.
- A "cross-language" interface for generic SQL features which were already natively handled in all targeted programming languages.

What makes Jakarta Commons more trustworthy than my own team? Why should I trust Sun's Java library over Microsoft's Java library? Or, more to the point....

4. Do we need loose coupling in Sakai?

After all, Sakai is meant to be the new de facto standard for open source LMS / CMS / CLE development. Why not just call all its services directly?

An evidence-based way to decide might be to look for symptoms which could be caused by overly tight coupling.

- **Unrealistic goals** - In the first Sakai all-hands meeting I attended, I heard someone say, "The problem is we need to do it right away and we need to do it right the first time." This is not a solvable problem. If you need to do something right away, you also need to plan on changing it. Loose coupling encourages that; tight coupling discourages it.
- **Slow refactoring** - Because every place that calls legacy code needs to be changed when the legacy interfaces change, and because direct calls make it harder to create unit tests, there's incentive not to make basic changes to the framework. As a result, high-level functional limitations and performance and maintenance issues linger on. Some Sakai services still use XML strings for their

data storage. Other tables still indicate relationships using embedded strings instead of foreign keys. Site set-up and membership management – key features of any collaborative web framework – are still combined in a single bristling UI and a single 12,000 line Java class.

- **"Living fossils"** - Because of the refactoring difficulties, what looks like the same capability may show up in multiple implementations in the same release. This results in extra bulk, extra documentation, extra QA, and a great deal of developer confusion. In Sakai 2.1, the hapless developer will find User, Agent, and SakaiPerson all providing user information, and come across CourseManagementService and CourseManagementManager in different packages.
- **Unpredictable disruptions** - During each Sakai release cycle I've witnessed, previously undocumented changes were made to framework code in the last few weeks of development. Every place that was tightly coupled to those areas of the framework would have to be changed and tested at a time when most projects have their own high-priority issues to deal with.
- **Non-standard packaging** - In 2005, Sakai began moving towards more standard industry practices, but its component system still mandates that Hibernate database mappings be split out of the standard application WAR archive and into a shared JAR. This causes eccentric source directory build structures and deployments, and conflicts with behavior-rich object design.

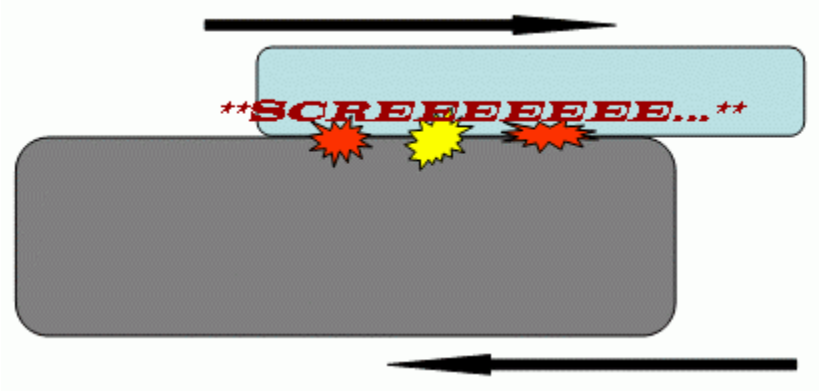
But even without this evidence, we would have used loose coupling.

Anyone who downloaded the first release of Sakai noticed that it seemed hefty for a 1.0 product. That's because Sakai didn't start simple and become more complicated one step at a time as needed. Instead, it started from U. Michigan's home-grown "CHEF" LMS, a large and complex application with its own long history.

Despite its size, CHEF (and therefore Sakai 1.0) didn't include some cross-institution requirements for a best-of-breed LMS/CMS/CLE service framework, presentation framework, and application suite. Release by release, more missing pieces have been brought in. There are still many gaps and problems, however.

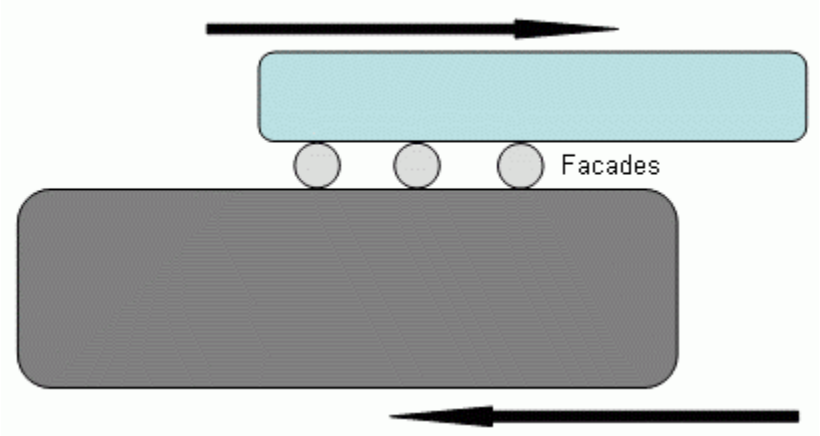
That's understandable, but it put us in a difficult spot: We needed to produce applications that depended on framework functionality which wasn't there yet. That meant there was no question of direct integration with a known stable interface: the interfaces weren't even defined and implemented.

We therefore had to coordinate two projects in motion with different goals and different teams:



5. Facades

When large moving objects interface, the key is to reduce inertial drag by reducing the amount of surface contact:



A *facade* is just an application-tailored interface to complex or unstable services. The difference between a "facade" and a plain old "service interface" is that the facade is designed specifically for the convenience of the particular client. The application, rather than the service, owns the code.

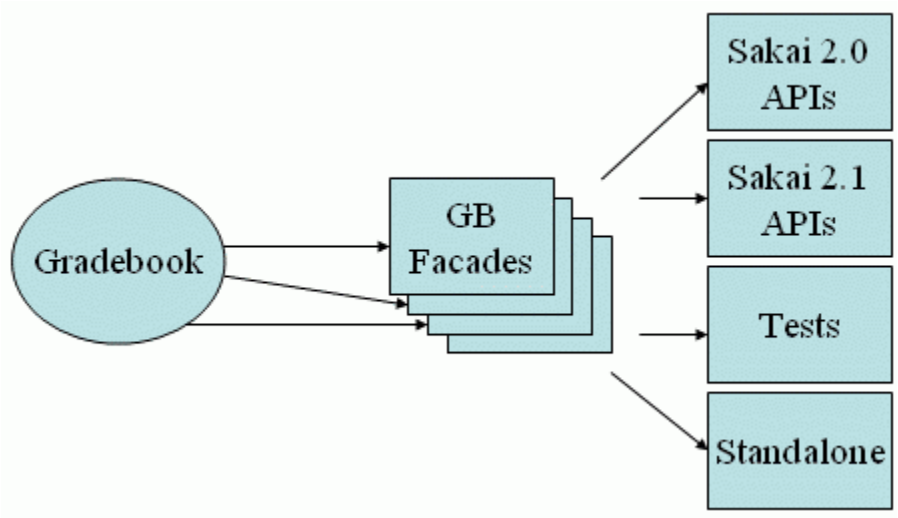
Successful facades:

- **Minimize maintenance costs** - When changes are made to a service interface, only one piece of application code needs to be changed. And, by using standalone implementations of the interfaces, the application and service projects become able to progress independently rather than interrupting each other at transition points or with bugs.
- **Reduce costs of unit and application testing** - To put it bluntly, you can't have unit tests without units. Keeping the interfaces as simple as possible makes "stubbing" easier – otherwise there may be just too many interfaces to mock up. At the application level, being able to build standalone speeds up the compile-and-test cycle and makes it easier to track down application-specific bugs and performance issues.

- **Self-document integration requirements** - Since facade interfaces show exactly what we need – no more, no less – it's easy for service developers and integrators to see what has to be delivered.
- **Maximize pluggability** - The job of plugging an application into a framework is simpler when the integration interfaces are centralized. Sakai's value is in potential delivery of best-of-breed open source applications to higher education. If a school like U. Washington or Harvard, which already has an institutional LMS in place, can find a reason to invest in OSP or our online assessments engine, that's a win for Sakai. Why make it more difficult than we have to?

In Java projects, facades are often used to keep business logic from being cluttered by specifics of JDBC, EJB, SOAP or other services, and Spring has quickly become a popular way to hide the peculiarities of the chosen technology from application-specific code.

5.1. Real-world example: Gradebook facades



Well-designed facades tend to be lightweight, which is why we can easily afford multiple implementations. When you look at what a web application really needs from a collaborative framework, it's usually not much. Many useful LAMP applications, for example, can get by with nothing more than authentication, and the majority can "plug into" a framework with just a context service and some external authorization code.

An online gradebook, by its very nature, is more closely tied to the specifics of higher education than most collaborative web applications would be, and therefore it requires more from a LMS/CMS framework. Even so, the 2.0 and 2.1 Gradebook facades are restricted to five areas:

- Authentication
- Context
- Authorization
- User Directory
- Site and Group Memberships and Roles

And most of these are very simple.

Authentication, for example, usually requires only one method: "Who is this?" (Or, more precisely, "How do I distinguish this user in the database and when talking to other services?")

```
public interface Authn {
   /**
    * @return an ID uniquely identifying the currently
    * authenticated user in a site, or null if the user
    * has not been authenticated.
    */
   public String getUserUid();
```

Context also only provides one piece of information: "Where am I?" (Or, more precisely, "How do I distinguish the chunk of data for the current site or class from the chunk of data which belongs to some other site or class?")

```
public interface ContextManagement {
   /**
    * @param request
    *    the javax.servlet.http.HttpServletRequest or
    *    javax.portlet.PortletRequest from which to determine the
    *    current gradebook. Since they don't share an interface,
    *    a generic object is passed.
    *
    * @return
    *    the UID of the currently selected gradebook, or null if the
    *   context manager cannot determine a selected gradebook
    */
   public String getGradebookUid(Object request);
```

Although **authorization** requirements tend to be more complicated by nature, we can still try to keep the complexity under control. In designing an authorization facade, we think pragmatically and concentrate on user workflows rather than on the finest grained objects and actions that are theoretically possible. Every combination of every externalized permission needs to be tested, maintained, somehow handled by the UI, and dealt with by the administrator. The fewer externalized permissions, the better for everyone.

```
public interface Authz {
   public boolean isUserAbleToGrade(String gradebookUid);
   public boolean isUserAbleToGradeAll(String gradebookUid);
   public boolean isUserAbleToGradeSection(String sectionUid);
   public boolean isUserAbleToEditAssessments(String gradebookUid);
   public boolean isUserAbleToViewOwnGrades(String gradebookUid);
   ...
```

As mentioned earlier, Sakai framework interfaces may undergo revision in the last few weeks before a release. Every time they change, all the code which calls them has to change. To meet my own delivery dates, I have a duty to keep those interruptions as contained as possible.

In this case, in the last week before feature freeze, the Gradebook's authorization approach changed from being roles-based to being based mostly on externally administered finer-grained-permissions (with site group permissions staying role-based). Also, the framework changed the way in which applications were expected to register their permission lists.

Here's the single file which handled both of these changes:

```
/**
 * An implementation of Gradebook-specific authorization needs based
 * on a combination of fine-grained site-scoped Sakai permissions and the
 * shared Section Awareness API. This is a transtional stage between
 * coarse-grained site-and-role-based authz and our hoped-for fine-grained
 * role-determined group-scoped authz.
 */
public class AuthzSakai2Impl extends AuthzSectionsImpl implements Authz {
   public static final String
      PERMISSION_GRADE_ALL = "gradebook.gradeAll",
      PERMISSION_GRADE_SECTION = "gradebook.gradeSection",
      PERMISSION_EDIT_ASSIGNMENTS = "gradebook.editAssignments",
      PERMISSION_VIEW_OWN_GRADES = "gradebook.viewOwnGrades";

   /**
    * Perform authorization-specific framework initializations for the Gradebook.
    */
   public void init() {
      FunctionManager.registerFunction(PERMISSION_GRADE_ALL);
      FunctionManager.registerFunction(PERMISSION_GRADE_SECTION);
      FunctionManager.registerFunction(PERMISSION_EDIT_ASSIGNMENTS);
      FunctionManager.registerFunction(PERMISSION_VIEW_OWN_GRADES);
   }

   public boolean isUserAbleToGrade(String gradebookUid) {
       return (hasPermission(gradebookUid, PERMISSION_GRADE_ALL) ||
          hasPermission(gradebookUid, PERMISSION_GRADE_SECTION));
   }

   /**
    * When group-scoped permissions are available, this is where
    * they will go. My current assumption is that the call will look like:
    *
    *   return hasPermission(sectionUid, PERMISSION_GRADE_ALL);
    */
   public boolean isUserAbleToGradeSection(String sectionUid) {
      return getSectionAwareness().isSectionMemberInRole(sectionUid, getAuthn().getUserUid(),
         Role.TA);


   ...
   }
```

At first, I was tempted to register permissions in some sort of core Sakai initialization class. But that would unnecessarily mix multiple concerns. Instead, by using Spring's bean initialization ability, we were able to keep all permissions-related code in one tidy package.

The application UI and business logic code stayed exactly the same.

5.2. What happens when a facade gets too thick?

The facade approach isn't a religion. It's a purely practical matter. And, as a practical matter, if a interface starts to become unusually complex, or if it starts to be shared between multiple projects, you'll want to rethink it.

In our case, membership data like groups and roles became much more complex when we added support for course sections. Moreover, other projects (like Tests & Quizzes) needed the same complex data. By negotiating an interface that would meet all our needs, we were able to pull what would have been duplicated work out of our individual projects and into a shared area: the "Section Awareness" service.

6. Loose coupling and project management

Risk identification is important in keeping a project under control. Once identified, risks might be prioritized and moved up in the schedule (so that estimates can become more reliable) or we might put a fallback position in place in case they explode.

Cross-project dependency points are inherently risky. Therefore, facade and service interfaces should be defined very early in the development cycle. Ideally, they're the first "final" code checked into the project.

This gives other projects more warning and more flexibility in making their own schedules. Projects which will rely on your services can mock up the agreed-upon interfaces for their own development and testing long before your team is finished with full implementation. Projects which need to provide services to you will have a clear explanation of what's needed from them.

Most of all, it gives your own project early warning. Even with the best intentions, it's always possible that an external dependency won't be met. Programmers may be overconfident or lack experience with firm deadlines; key team members are transferred; groups get downsized; priorities change....

For example, early in the Sakai 2.0 Gradebook project, previous assurances of a functionality-driven schedule were reversed and we were told that we had to deliver in four months, no matter what. In this case, we scaled back our facades (and the features which depended on them) to match what the existing framework was able to support. This flexibility was possible because user representatives were full members of the development team.

In such circumstances, it may be possible for the team itself to implement missing parts of the external dependency, assuming the work can fit into the schedule. In more modular areas of Sakai, such as the JSF components, this may be a fairly simple matter. In closed or dangerously complex territory, such as the core framework or legacy services, a "middleware" approach may be able to serve as a fallback when direct dependencies aren't met, albeit at the possible cost of creating redundancies or "living fossils".

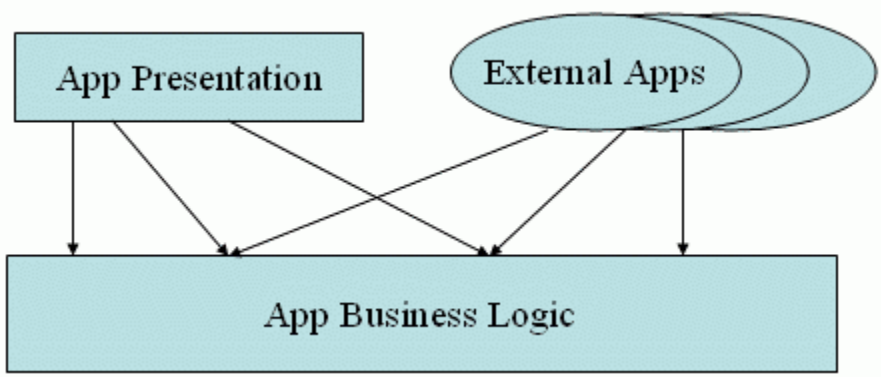
The Sakai 2.1 section-aware projects required site membership data which wasn't part of the Sakai 2.0 core framework. We therefore used standalone facade implementations to enable our development and testing before the framework was updated. Those same "middleware" implementations could fairly easily have been used for the actual 2.1 release if necessary. Either way, application-specific code didn't have to change.

7. Loosely coupled applications

There are two sides to loose coupling. Most commonly, an application developer is a **consumer of services**. As a consumer, we achieve loose coupling through Spring-injected facades.

But we may also sometimes be a **producer of services** to be used by other applications.

Application = Tool + Component?



One way to approach service provision is to think of your application as a big chunk of open business logic, with your UI just one client among many. The appealing thing about this idea is that it looks so efficient. All you do is separate presentation code from the business model, and then everything will be re-usable with no extra work!

But that's true only so long as nothing changes. And in empirical programming we explicitly *count* on things changing.

I've worked on operating system run-time libraries and data management packages. I've worked on multimedia authoring tools and real-time graphic analysis. MVC design or not, delivering a shared service and delivering a web application are *not the same job*.

You collect different requirements from different stakeholders. You have different tests. You produce different documentation. An application typically needs previews and multi-step processes; a service typically needs efficient bulk transactions. As simple as it was, the Sakai 2.0 Gradebook needed to apply different authorization rules for the application and the service: a student can't change their own grades through the application, but when a student submits answers in the Tests & Quizzes tool, they'll be changing their own grade. Even when an application and a shared service are developed by the same team and end up in the same database tables, they're logically different tasks.

If I pretend they're the same, I do both types of customer a disservice. I'll be reluctant to respond to user needs in an agile fashion because of the possibility of breaking external callers, but I'm almost certain to break external callers eventually because application needs come first.

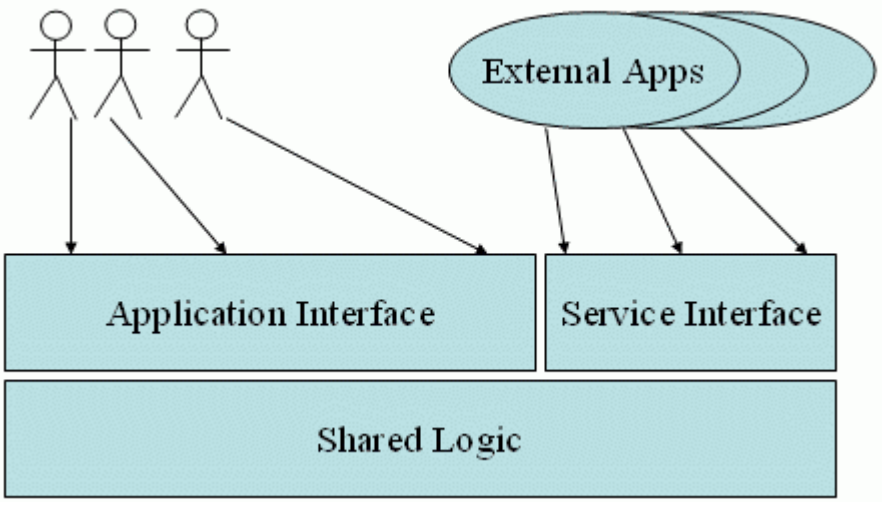
	Application	Service
Customers	Instructors; students	Programmers
Goals	Usable browser-based workflow	Efficient integration
Contracts	Functional specification; wireframes; prototypes	API; unit tests
Project lifecycles	Rapid change	Negotiated stability

Probably the most successful Java example of combining a usable application with a pluggable framework is the Eclipse project. Erich Gamma, a member of the *Design Patterns* "Gang of Four" and co-developer of JUnit, also helps lead Eclipse. A while back, I ran into an interview with Gamma on the topic of re-use:

> You can go and expose everything, and people can change anything. The problems start when the next version comes along. If you have exposed everything, you cannot change anything or you break all your clients. APIs don't just happen; they are a big investment. ... I really like flexibility that's requirement driven. That's also what we do in Eclipse. When it comes to exposing more API, we do that on demand. We expose API gradually.... So I really think about it in smaller steps, we do not want to commit to an API before its time.

Instead of pretending we'll get something for nothing, we prefer to explicitly separate the deliverables:

Project = Application + Service



An API is a contract. You don't want to enter into a contract that you can't honor or that you can't realistically expect to be honored by the other party.